# 赵凯笔记-PIMPL

# 什么是PIMPL？

PIMPL是指pointer to implementation。通过使用**指针的方式隐藏对象的实现细节**。是实现“将文件间的编译依存关系降至最低”的方法之一。另一个方式是通过接口实现，但其原理一样。

PIMPL又称作“编译防火墙”、“笑脸猫技术”，它只在C/C++等编译语言中起作用。

# 为什么要使用PIMPL？

## 理论分析

庞大的项目，修改一个文件之后，重新编译，所有依赖该文件的文件都需要重新编译，导致编译时间太长。

## 工程实例

通过描述一个实例来证明上一小节的理论。

### 不使用PIMPL

文件间的依赖关系如图：



有三个源文件依赖“Person.h”，实际中可以有更多个文件依赖它，为了说明意思，我源码写的都非常简单，主要是为了表明文件间的依赖关系而已。

**Person.h**

#ifndef PERSON\_H\_

#define PERSON\_H\_

struct Person

{

void print();

};

#endif

**Person.cc**

#include "Person.h"

#include <iostream>

void

Person::print()

{

std::cout << "Person::print()" << std::endl;

}

**PersonUser.cc**

#include "Person.h"

**main.cc**

#include "Person.h"

int main()

{

return 0;

}

**Makefile**

#

# Makefile

# author: zhaokai

# date: 2013-11-28

#

TESTS = main

all : $(TESTS)

clean :

rm -f $(TESTS)

rm -f main.o PersonUser.o Person.o

main.o: main.cc Person.h

g++ -c main.cc

PersonUser.o : PersonUser.cc Person.h

g++ -c PersonUser.cc

Person.o: Person.cc Person.h

g++ -c Person.cc

$(TESTS): main.o PersonUser.o Person.o

g++ -o main main.o PersonUser.o Person.o

现在我们开始修改Person.h文件：

#ifndef PERSON\_H\_

#define PERSON\_H\_

struct Person

{

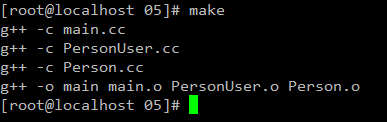
int i; // add int i

void print();

};

#endif

然后make，结果如下：



依赖Person.h的三个文件都被重新编译了，最后链接生成执行文件。

### 使用PIMPL

使用PIMPL需要将Person类的实现移到PersonImpl类中，使用指针的方式将实现隐藏，相当于Person.h只是一个傀儡而已，而以前依赖它的文件依旧依赖之，文件间的依赖关系如图：



有三个源文件依赖“Person.h”，实际中可以有更多个文件依赖它，有两个源文件和一个头文件依赖“PersonImpl.h”。

**Person.h**

#ifndef PERSON\_H\_

#define PERSON\_H\_

#include <memory>

struct PersonImpl;

struct Person

{

void print();

private:

std::shared\_ptr<PersonImpl> pImpl;

};

#endif

**Person.cc**

#include "Person.h"

#include "PersonImpl.h"

void

Person::print()

{

pImpl->print();

}

**PersonImpl.h**

#ifndef PERSONIMPL\_H\_

#define PERSONIMPL\_H\_

struct PersonImpl

{

void print();

};

#endif

**PersonImpl.cc**

#include "PersonImpl.h"

#include <iostream>

void

PersonImpl::print()

{

std::cout << "PersonImpl::print()" << std::endl;

}

**PersonUser.cc**

#include "Person.h"

**main.cc**

#include "Person.h"

int main()

{

return 0;

}

**Makefile**

#

# Makefile

# author: zhaokai

# date: 2013-11-28

#

TESTS = main

all : $(TESTS)

clean :

rm -f $(TESTS)

rm -f main.o PersonUser.o Person.o PersonImpl.o

main.o: main.cc Person.h

g++ --std=c++11 -c main.cc

PersonUser.o : PersonUser.cc Person.h

g++ --std=c++11 -c PersonUser.cc

Person.o: Person.cc Person.h PersonImpl.h

g++ --std=c++11 -c Person.cc

PersonImpl.o: PersonImpl.cc PersonImpl.h

g++ --std=c++11 -c PersonImpl.cc

$(TESTS): main.o PersonUser.o Person.o PersonImpl.o

g++ -o main main.o PersonUser.o Person.o PersonImpl.o

现在我们开始修改PersonImpl.h文件，注意这时候Person.h已经是傀儡了，如果想给Person增加属性那应该修改PersonImpl.h文件：

#ifndef PERSONIMPL\_H\_

#define PERSONIMPL\_H\_

struct PersonImpl

{

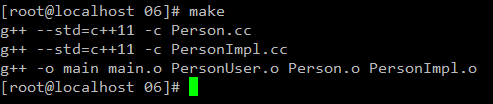
int i; // add int i

void print();

};

#endif

然后make，结果如下：



依赖PersonImpl.h的两个文件都重新编译了，而依赖于“Person.h”的文件main.cc和PersonUser.cc都没有重新编译。

### 对比

同样是一件事情，为Person类增加属性int i；两种方法导致编译的过程就不同，我们举得例子比较小，如果有100个类似PersonUser这样的文件，那么使用PIMPL，编译时还是只有“Person.cc”和“PersonImpl”两个文件重新编译了；但是不使用PIMPL的话，就是“main.cc”，“Person.cc”和100个类似“PersonUser.cc”这样的文件重新编译，那就是102个文件。

通过上面的实例就可证明理论分析部分了。

# 如何使用PIMPL

有多种方式实现PIMPL，这里按照《Effective C++》中介绍的方式。

## 基本步骤

假设原有Person如下：

Person.h

struct **Person**

**{**

public**:**

**Person(**const int \_age**);**

void print**();**

private**:**

int age**;**

**};**

Person.cc

Person**::**Person**(**const int \_age**)**

**:** age**(**\_age**)**

**{}**

void

Person**::**print**()**

**{**

std**::**cout **<<** "Person::print::age=" **<<** age **<<** std**::**endl**;**

**}**

### 将Person改名为PersonImpl

PersonImpl.h

struct **PersonImpl**

**{**

public**:**

**PersonImpl(**const int \_age**);**

void print**();**

private**:**

int age**;**

**};**

PersonImpl.cc

**PersonImpl::PersonImpl(**const int \_age**)**

**:** age**(**\_age**)**

**{}**

void

**PersonImpl::**print**()**

**{**

std**::**cout **<<** "**PersonImpl**::print::age=" **<<** age **<<** std**::**endl**;**

**}**

### 抽象public和protected方法

将PersonImpl中的public和protected方法成Person。Person中的方法实际调用的是PersonImpl中对应的方法，Person的定义中需要使用PersonImpl，在Person.h文件中绝对不能#include “PersonImpl.h”，这样就是做无用功了。

* **为什么不能将PersonImpl的对象作为Person的属性？**

因为Person类的定义中需要知道该类的大小，如果直接使用PersonImpl的对象，那么就必须知道PersonImpl的定义，而我们恰恰希望在Person的定义中隐藏PersonImpl的定义。这时候，指针就大显神通了。因为指针的大小只与操作系统的位数有关（32位的机器都占4个字节，64位的机器都占8个字节），所以我们可以使用指针指向PersonImpl，从而只需要前置声明就可以了。这就是暗度陈仓吧。

Person.h

struct **PersonImpl;** **// 前置声明，PIMPL的关键**

struct Person

**{**

public**:**

Person**(**const int \_age**);**

void print**();**

private**:**

std**::**shared\_ptr**<**PersonImpl**>** pImpl**;**  **// 指针，暗度陈仓**

**};**

Person.cc

Person**::**Person**(**const int \_age**)**

**:** pImpl**(new** PersonImpl**(**\_age**))**

**{}**

void

Person**::**print**()**

**{**

pImpl**->**print**();**

**}**

### 使用Person

在其它地方我们就可以使用Person了，例如：

int main**()**

**{**

Person p**(**1**);**

p**.**print**();**

**return** 0**;**

**}**

### 示例源码下载

# 优缺点

## 优点

* 改变类的私有成员无需重新编译依赖它的文件，所以整个工程重新编译的速度会快很多。
* 头文件中采用声明式（如下所示，其中PersonImpl的声明就是采用声明式），因此编译时间会快很多。

**Person.h**

#ifndef PERSON\_H\_

#define PERSON\_H\_

#include <memory>

**struct PersonImpl**;

struct Person

{

void print();

private:

std::shared\_ptr<PersonImpl> pImpl;

};

#endif

## 缺点

* 实现者需要做更多的工作
* 代码会变得不易理解
* Run-time performance is slightly compromised due to the pointer indirection, especially if function calls are virtual (branch prediction for indirect branches is generally poor).

# 参考

1. 《effective C++》 条款31：将文件间的编译关系降至最低
2. PIMPL Idiom： <http://c2.com/cgi/wiki?PimplIdiom>