零声教育出品 Mark 老师 QQ: 2548898954

单例模式

定义

> 保证一个类仅有一个实例，并提供一个该实例的全局访问点。
> ——《设计模式》GoF

版本一

```
class Singleton {
public:
    static Singleton * GetInstance() {
        if (_instance == nullptr) {
            _instance = new Singleton();
        }
        return _instance;
    }
private:
    Singleton(){}; //构造
    ~Singleton(){};
    Singleton(const Singleton &) = delete; //拷贝构造
    Singleton& operator=(const Singleton&) = delete;//拷贝赋值构造
    Singleton(Singleton &&) = delete;//移动构造
    Singleton& operator=(Singleton &&) = delete;//移动拷贝构造
    static Singleton * _instance;
};
```

```
Singleton* Singleton::_instance = nullptr;//静态成员需要初始化

```

版本二

```
class Singleton {
public:
    static Singleton * GetInstance() {
        if (_instance == nullptr) {
            _instance = new Singleton();
            atexit(Destructor);
        }
        return _instance;
    }
private:
    static void Destructor() {
        if (nullptr != _instance) { //
            delete _instance;
            _instance = nullptr;
        }
    }
    Singleton(){}; //构造
    ~Singleton(){};
    Singleton(const Singleton &) = delete; //拷贝构造
    Singleton& operator=(const Singleton&) = delete;//拷贝赋值构造
    Singleton(Singleton &&) = delete;//移动构造
    Singleton& operator=(Singleton &&) = delete;//移动拷贝构造
    static Singleton * _instance;
};
Singleton* Singleton::_instance = nullptr;//静态成员需要初始化
```

```
// 还可以使用 内部类，智能指针来解决； 此时还有线程安全问题
```

版本三

```
#include <mutex>
class Singleton { // 懒汉模式 lazy load
public:
    static Singleton * GetInstance() {
        // std::lock_guard<std::mutex> lock(_mutex); // 3.1 切换线程
        if (_instance == nullptr) {
            std::lock_guard<std::mutex> lock(_mutex); // 3.2
            if (_instance == nullptr) {
                _instance = new Singleton();
                // 1. 分配内存
                // 2. 调用构造函数
                // 3. 返回指针
                // 多线程环境下 cpu reorder操作
                atexit(Destructor);
            }
        }
        return _instance;
    }
private:
    static void Destructor() {
        if (nullptr != _instance) {
            delete _instance;
            _instance = nullptr;
        }
    }
    Singleton(){}; //构造
    ~Singleton(){};
    Singleton(const Singleton &) = delete; //拷贝构造
```

```
    Singleton& operator=(const Singleton&) = delete;//拷贝赋值构造
    Singleton(Singleton &&) = delete;//移动构造
    Singleton& operator=(Singleton &&) = delete;//移动拷贝构造
    static Singleton * _instance;
    static std::mutex _mutex;
};
Singleton* Singleton::_instance = nullptr;//静态成员需要初始化
std::mutex Singleton::_mutex; //互斥锁初始化
```

版本四

```
// volitile
#include <mutex>
#include <atomic>
class Singleton {
public:
    static Singleton * GetInstance() {
        Singleton* tmp = _instance.load(std::memory_order_relaxed);

 std::atomic_thread_fence(std::memory_order_acquire);//获取内存屏障
        if (tmp == nullptr) {
            std::lock_guard<std::mutex> lock(_mutex);
            tmp = _instance.load(std::memory_order_relaxed);
            if (tmp == nullptr) {
                tmp = new Singleton;

 std::atomic_thread_fence(std::memory_order_release);//释放内存屏障
```

```
                _instance.store(tmp,
std::memory_order_relaxed);
                atexit(Destructor);
            }
        }
        return tmp;
    }
private:
    static void Destructor() {
        Singleton* tmp =
_instance.load(std::memory_order_relaxed);
        if (nullptr != tmp) {
            delete tmp;
        }
    }
    Singleton(){}; //构造
    ~Singleton(){};
    Singleton(const Singleton &) = delete; //拷贝
构造
    Singleton& operator=(const Singleton&) =
delete;//拷贝赋值构造
    Singleton(Singleton &&) = delete;//移动构造
    Singleton& operator=(Singleton &&) =
delete;//移动拷贝构造
    static std::atomic<Singleton*> _instance;
    static std::mutex _mutex;
};
std::atomic<Singleton*> Singleton::_instance;//静
态成员需要初始化
std::mutex Singleton::_mutex; //互斥锁初始化
// g++ Singleton.cpp -o singleton -std=c++11
```

版本五

```
// c++11 magic static 特性：如果当变量在初始化的时候，
并发同时进入声明语句，并发线程将会阻塞等待初始化结束。
```

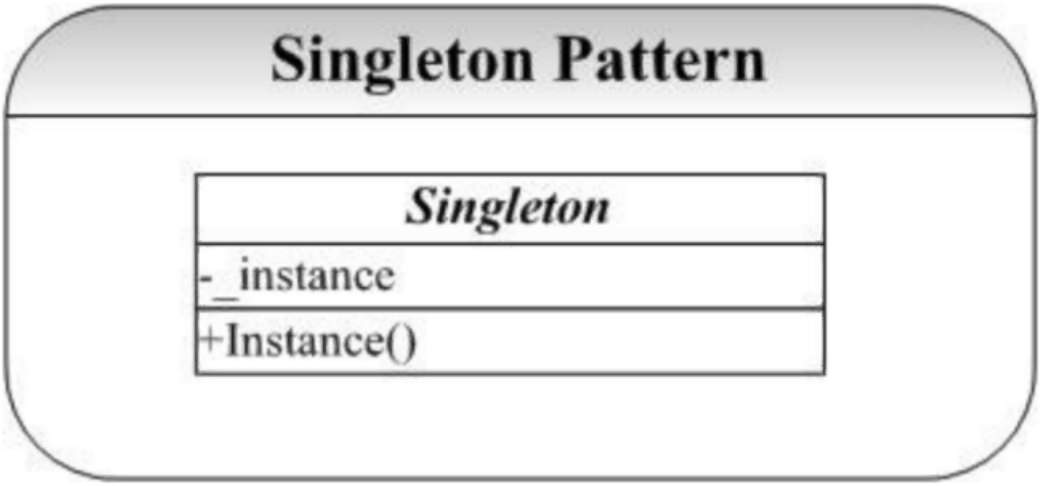
```
// c++ effective

class Singleton
{
public:
    static Singleton& GetInstance() {
        static Singleton instance;
        return instance;
    }
private:
    Singleton(){}; //构造
    ~Singleton(){};
    Singleton(const Singleton &) = delete; //拷贝构造
    Singleton& operator=(const Singleton&) = delete;//拷贝赋值构造
    Singleton(Singleton &&) = delete;//移动构造
    Singleton& operator=(Singleton &&) = delete;//移动拷贝构造
};
// 继承 Singleton
// g++ Singleton.cpp -o singleton -std=c++11
/*该版本具备 版本5 所有优点：
1. 利用静态局部变量特性，延迟加载；
2. 利用静态局部变量特性，系统自动回收内存，自动调用析构函数；
3. 静态局部变量初始化时，没有 new 操作带来的cpu指令reorder操作；
4. c++11 静态局部变量初始化时，具备线程安全；
*/
```

版本六

```
template<typename T>
class Singleton {
public:
```

```
    static T& GetInstance() {
        static T instance; // 这里要初始化DesignPattern，需要调用DesignPattern 构造函数，同时会调用父类的构造函数。
        return instance;
    }
protected:
    virtual ~Singleton() {}
    Singleton() {} // protected修饰构造函数，才能让别人继承
private:
    Singleton(const Singleton &) = delete; //拷贝构造
    Singleton& operator=(const Singleton&) = delete;//拷贝赋值构造
    Singleton(Singleton &&) = delete;//移动构造
    Singleton& operator=(Singleton &&) = delete;//移动拷贝构造
};
class DesignPattern : public Singleton<DesignPattern> {
    friend class Singleton<DesignPattern>; // friend 能让Singleton<T> 访问到 DesignPattern构造函数
private:
    DesignPattern() {}
    ~DesignPattern() {}
};
```

结构图



工厂方法

定义

> 定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method使得一个类的实例化延迟到子类。 ——《设计模式》GoF

背景

> 实现一个导出数据的接口，让客户选择数据的导出方式；

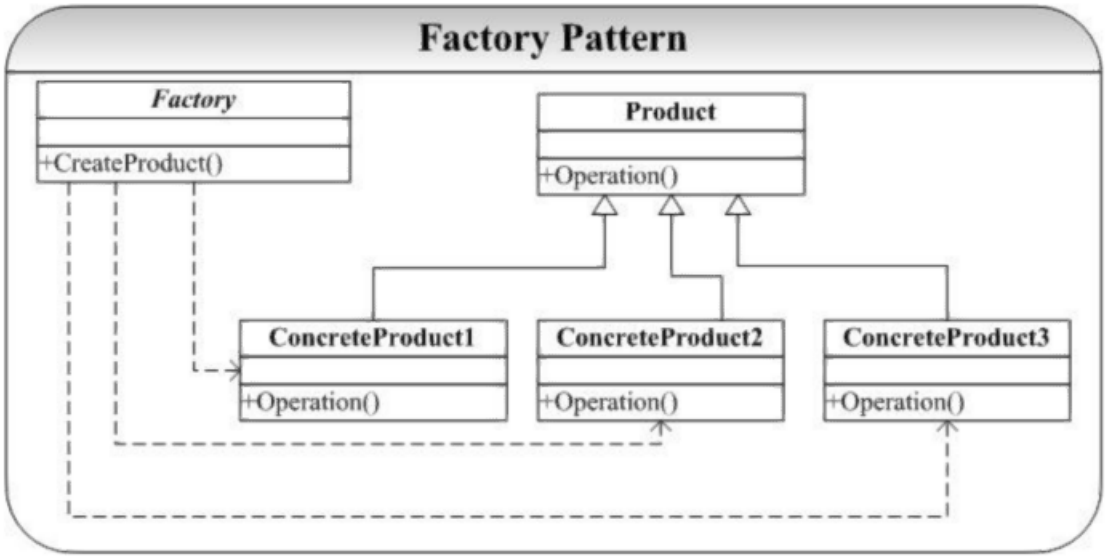
要点

> 解决创建过程比较复杂，希望对外隐藏这些细节的场景；
>
> - 比如连接池、线程池
> - 隐藏对象真实类型；
> - 对象创建会有很多参数来决定如何创建；
> - 创建对象有复杂的依赖关系；

本质

- 延迟到子类来选择实现；

结构图



抽象工厂

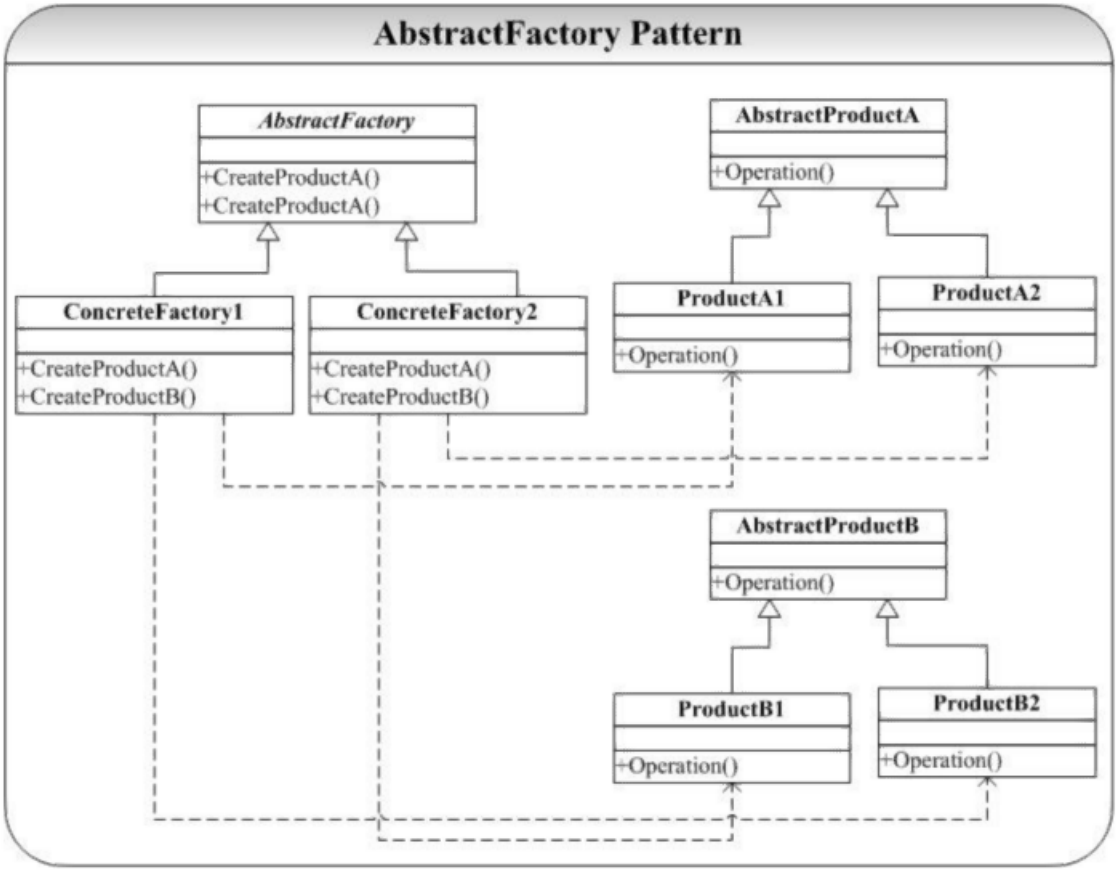
定义

> 提供一个接口，让该接口负责创建一系列“相关或者相互依赖的对象”，无需指定它们具体的类。 ——《设计模式》GoF

背景

> 实现一个拥有导出导入数据的接口，让客户选择数据的导出导入方式；

结构图



责任链

定义

> 使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递请求，直到有一个对象处理它为止。 ——《设计模式》GoF

背景

> 请求流程，1 天内需要主程序批准，3 天内需要项目经理批准，3 天以上需要老板批准；

要点

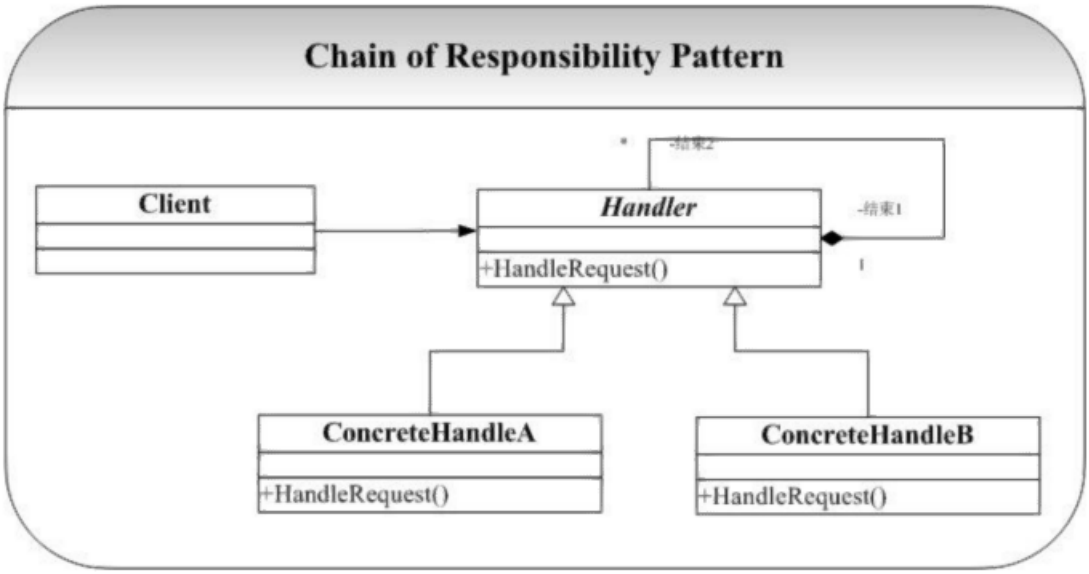
- 解耦请求方和处理方，请求方不知道请求是如何被处理，处理方的组成是由相互独立的子处理构成，子处理流程通过链表的方式连接，子处理请求可以按任意顺序组合；

- 责任链请求强调请求最终由一个子处理流程处理；通过了各个子处理条件判断；
- 责任链扩展就是功能链，功能链强调的是，一个请求依次经由功能链中的子处理流程处理；
- 将职责以及职责顺序运行进行抽象，那么职责变化可以任意扩展，同时职责顺序也可以任意扩展；

本质

- 分离职责，动态组合；

结构图

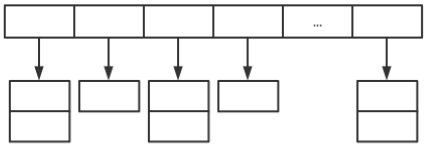


nginx 阶段处理

结构图

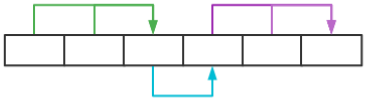
ngx_http_phase_t phases[NGX_HTTP_LOG_PHASE + 1];

```
typedef ngx_int_t (*ngx_http_handler_pt)(ngx_http_request_t *r);
typedef struct {
    ngx_array_t                handlers;
} ngx_http_phase_t;
```

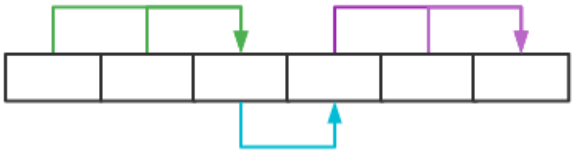


ngx_http_phase_engine_t phase_engine;

```
typedef struct {
    ngx_http_phase_handler_pt  checker;
    ngx_http_handler_pt        handler;
    ngx_uint_t                 next;
}ngx_http_phase_handler_t;

typedef struct {
    ngx_http_phase_handler_t  *handlers;
    ngx_uint_t                 server_rewrite_index;
    ngx_uint_t                 location_rewrite_index;
} ngx_http_phase_engine_t;
```



调度图



```
ph = cmcf->phase_engine.handlers;
while (ph[r->phase_handler].checker) {
    rc = ph[r->phase_handler].checker(r, &ph[r->phase_handler]);
    if (rc == NGX_OK) {
        return;
    }
}
```

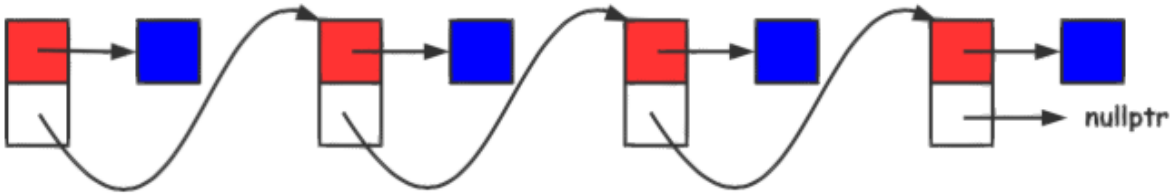
装饰器

定义

> 动态地给一个对象增加一些额外的职责。就增加功能而言，装饰器模式比生产子类更为灵活。 —— 《设计模式》GoF

背景

> 普通员工有销售奖金，累计奖金，部门经理除此之外还有团队奖金；后面可能会添加环比增长奖金，同时可能针对不同的职位产生不同的奖金组合；



要点

- 通过采用组合而非继承的手法， 装饰器模式实现了在运行时动态扩展对象功能的能力，而且可以根据需要扩展多个功能。 避免了使用继承带来的“灵活性差”和“多子类衍生问题”。
- 不是解决“多子类衍生问题”问题，而是解决“父类在多个方向上的扩展功能”问题；
- 装饰器模式把一系列复杂的功能分散到每个装饰器当中，一般一个装饰器只实现一个功能，实现复用装饰器的功能；

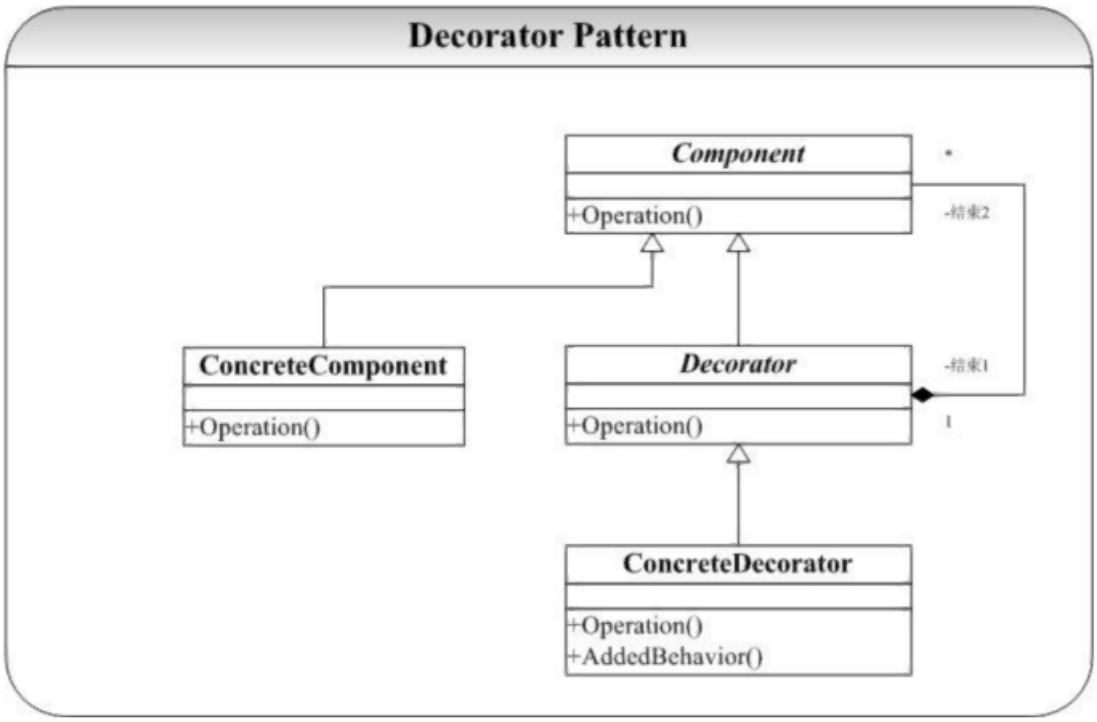
什么时候使用?

- 不影响其他对象的情况下，以动态、透明的方式给对象添加职责；每个职责都是完全独立的功能，彼此之间没有依赖；

本质

- 动态组合

结构图



组合模式

定义

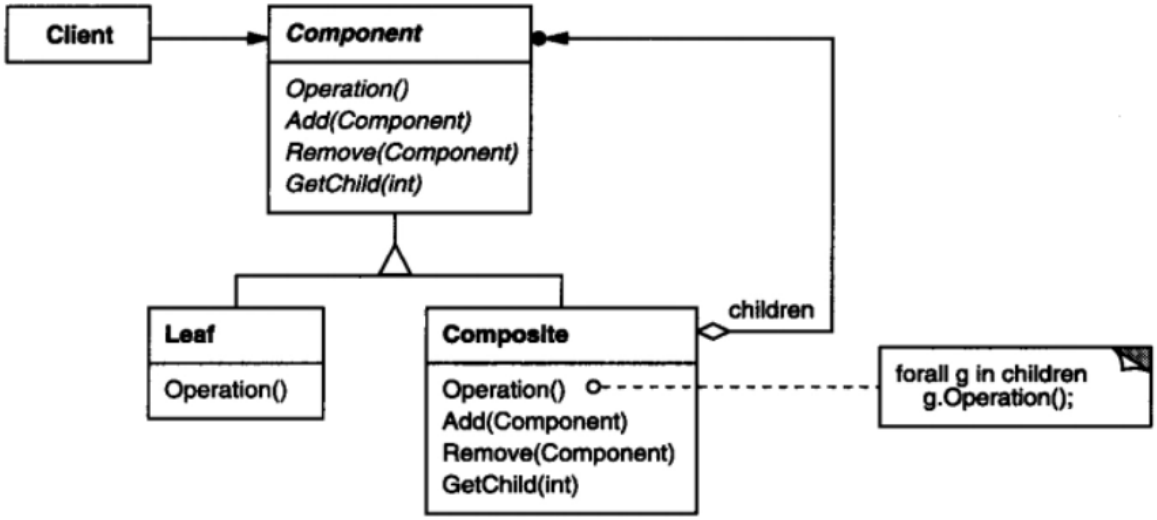
> 将对象组合成树型结构以表示“部分-整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

什么时候使用组合模式?

- 如果你想**表示对象的部分-整体层次结构**，可以选用组合模式，把整体和部分的操作统一起来，使得层次结构实现更简单，从外部来使用这个层次结构也容易；
- 如果你希望**统一地使用组合结构中的所有对象**，可以选用组合模式，这正是组合模式提供的主要功能；

怎么实现?

> 将叶子节点当成特殊的组合对象看待，从而统一叶子对象和组合对象；


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
forall g in children
g.Operation();