

Apply Orthogonal Design in C++

应用正交设计

刘光聪

liu.guangcong@zte.com.cn

2017-02-28

内容

1 软件设计概述

2 面向对象设计

3 函数式设计

4 文献

软件设计概述

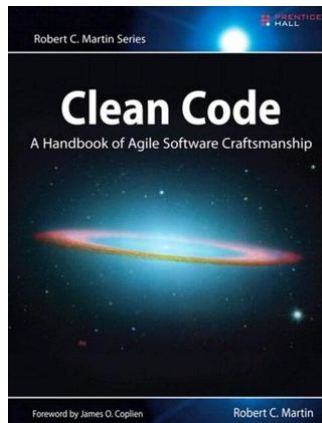
设计目标





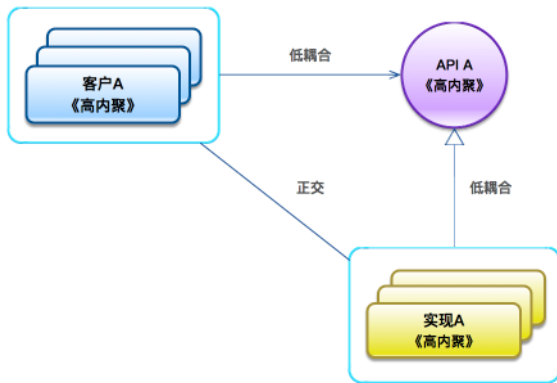
易于理解

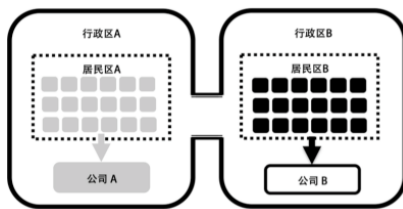
- 1 Clean Code
- 2 Idioms
- 3 Patterns



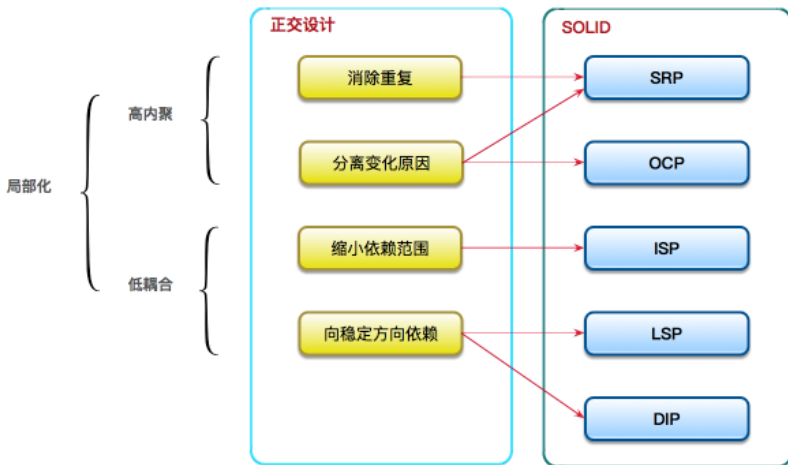
- 1 **YAGNI**: You Ain't Gonna Need It
- 2 **KISS**: Keep it Simple, Stupid

模块化设计





正交设计



拥抱变化

应对变化

- ① 一个变化导致多处散弹修改：消除重复
- ② 多个变化导致一处频繁修改：分离变化

变化发生时，消除不必要的修改

- ① 不依赖不必要的依赖：缩小依赖范围
- ② 不依赖不稳定的依赖：向着稳定的方向依赖

面向对象设计

迭代 1

需求

❶ 需求 1：判断某个单词是否包含数字

类型推演

```
bool hasDigit(const string& word) {  
    for (auto c : word)  
        if (isdigit(c))  
            return true;  
    return false;  
}
```

- 意图明确
- 关注性能
- 消除冗余噪声
- 增强编译时安全

需求

- ❶ 需求 1：判断某个单词是否包含数字
- ❷ 需求 2：判断某个单词是否包含大写字母

重复设计：复制 & 粘贴

```
bool hasUpper(const string& word) {  
    for (auto c : word)  
        if (isupper(c))  
            return true;  
    return false;  
}
```

- 消除重复
- 分离变化

DRY



抽象

```
template <typename Matcher>
bool exists(const string& word, Matcher matcher) {
    for (auto c : word)
        if (matcher(c))
            return true;
    return false;
}
```

- 愿意被第一颗子弹击中，但拒绝同一方向的子弹再次被击中
- 愚弄我一次，应感羞愧的是你；再次愚弄我，应感羞愧的是我
- 没有一种设计是永远 OCP，只存在对当前变化保持 OCP

泛化

```
template <typename Iterator, typename Matcher>
bool exists(Iterator first, Iterator last, Matcher matcher) {
    for (; first != last; ++first)
        if (matcher(*first))
            return true;
    return false;
}
```

- 容器类型
- 迭代算法
- 匹配准则

无参函数对象

```
struct {  
    bool operator()(char c) const {  
        return std::isdigit(c);  
    }  
} is_digit;  
  
struct {  
    bool operator()(char c) const {  
        return std::isupper(c);  
    }  
} is_upper;  
  
exists(word.begin(), word.end(), is_digit);  
exists(word.begin(), word.end(), is_upper);
```

- 组合小类
- 复用对象

迭代 3

需求

- ❶ 需求 1：判断某个单词是否包含数字
- ❷ 需求 2：判断某个单词是否包含大写字母
- ❸ 需求 3：判断某个单词是否包含下划线

有参函数对象

```
template <typename T>
struct Equals {
    explicit Equals(const T& t) : t(t)
    {}

    bool operator()(const T& t) const {
        return this->t == t;
    }

private:
    T t;
};

exists(word.begin(), word.end(), Equals<char>('_'));
```

● 心智包袱

工厂方法：equals_to

```
template <typename T>
auto equals_to(const T& t) {
    return Equals<T>(t);
}
```

```
exists(word.begin(), word.end(), equals_to('_'));
```

- 类型推演
- 零负担

实用对象

```
const Equals<std::string> is_empty("");  
const Equals<const void*> is_nil(nullptr);
```

- 语法糖
- 复用对象

需求

- ❶ 需求 1：判断某个单词是否包含数字
- ❷ 需求 2：判断某个单词是否包含大写字母
- ❸ 需求 3：判断某个单词是否包含下划线
- ❹ 需求 4：判断某个单词是否不包含下划线

修饰语义

```
template <typename Matcher>
struct Not {
    Not(const Matcher& matcher)
        : matcher(matcher)
    {}

    template <typename T>
    bool operator()(const T& actual) const {
        return !matcher(actual);
    }

private:
    Matcher matcher;
};
```

- 修饰语义
- 增强功能

工厂方法: is_not

```
template <typename Matcher>
auto is_not(const Matcher& matcher) {
    return Not<Matcher>(matcher);
}

forall(word.begin(), word.end(), is_not(equals_to('_')));
```

重复再现

```
template <typename Iterator, typename Matcher>
bool exists(Iterator first, Iterator last, Matcher matcher) {
    for (; first != last; ++first)
        if (matcher(*first))
            return true;
    return false;
}

template <typename Iterator, typename Matcher>
bool forall(Iterator first, Iterator last, Matcher matcher) {
    for (; first != last; ++first)
        if (!matcher(*first))
            return false;
    return true;
}
```

提取函数

```
template <bool shortcut, typename Iterator, typename Matcher>
bool expect(Iterator first, Iterator last, Matcher matcher) {
    for (; first != last; ++first)
        if (matcher(*first) == shortcut)
            return shortcut;
    return !shortcut;
}

template <typename Iterator, typename Matcher>
bool exists(Iterator first, Iterator last, Matcher matcher) {
    return expect<true>(first, last, matcher);
}

template <typename Iterator, typename Matcher>
bool forall(Iterator first, Iterator last, Matcher matcher) {
    return expect<false>(first, last, matcher);
}
```

需求

- ❶ 需求 1：判断某个单词是否包含数字
- ❷ 需求 2：判断某个单词是否包含大写字母
- ❸ 需求 3：判断某个单词是否包含下划线
- ❹ 需求 4：判断某个单词是否不包含 _
- ❺ 需求 5：判断某个单词是否包含 _，或者 *

组合或

```
template <typename LeftMatcher, typename RightMatcher>
struct Or {
    Or(const LeftMatcher& left, const RightMatcher& right)
        : left(left), right(right) {}

    template <typename T>
    bool operator()(const T& actual) const {
        return left(actual) || right(actual);
    }

private:
    LeftMatcher left;
    RightMatcher right;
};
```


迭代 5

组合或：工厂

```
template <typename LeftMatcher, typename RightMatcher>
auto is_or(const LeftMatcher& left, const RightMatcher& right) {
    return Or<LeftMatcher, RightMatcher>(left, right);
}

exists(word.begin(), word.end(), is_or(
    equals_to('_'), equals_to('*')
));
```

需求

- ❶ 需求 1: 判断某个单词是否包含数字
- ❷ 需求 2: 判断某个单词是否包含大写字母
- ❸ 需求 3: 判断某个单词是否包含下划线
- ❹ 需求 4: 判断某个单词是否不包含 _
- ❺ 需求 5: 判断某个单词是否包含 _，或者
- ❻ 需求 6: 判断某个单词是否包含空白符，但除去空格

组合与

```
template <typename LeftMatcher, typename RightMatcher>
struct And {
    And(const LeftMatcher& left, const RightMatcher& right)
        : left(left), right(right) {
    }

    template <typename T>
    bool operator()(const T& actual) const {
        return left(actual) && right(actual);
    }

private:
    LeftMatcher left;
    RightMatcher right;
};
```

工厂：组合与

```
template <typename LeftMatcher, typename RightMatcher>
auto is_and(const LeftMatcher& left, const RightMatcher& right) {
    return And<LeftMatcher, RightMatcher>(left, right);
}

exists(word.begin(), word.end(), is_and(
    is_space, is_not(equals_to(' '))
));
```

需求

- ❶ 需求 1: 判断某个单词是否包含数字
- ❷ 需求 2: 判断某个单词是否包含大写字母
- ❸ 需求 3: 判断某个单词是否包含下划线
- ❹ 需求 4: 判断某个单词是否不包含 _
- ❺ 需求 5: 判断某个单词是否包含 _，或者
- ❻ 需求 6: 判断某个单词是否包含空白符，但除去空格
- ❼ 需求 7: 判断某个单词是否包含字母 x，且不区分大小写

修饰

```
template <typename Matcher, typename T>
struct IgnoringCase {
    explicit IgnoringCase(const T& expect)
        : matcher(toLower(expect))
    {}

    bool operator()(const T& actual) const {
        return matcher(toLower(actual));
    }

private:
    Matcher matcher;
};
```

引入工厂

```
template <typename T>
auto ignoring_case_equals(const T& expected) {
    return IgnoringCase<Equals<T>, T>(expected);
}

exists(word.begin(), word.end(), ignoring_case_equals('x'));
```

需求

- ❶ 需求 1: 判断某个单词是否包含数字
- ❷ 需求 2: 判断某个单词是否包含大写字母
- ❸ 需求 3: 判断某个单词是否包含下划线
- ❹ 需求 4: 判断某个单词是否不包含 _
- ❺ 需求 5: 判断某个单词是否包含 _ , 或者
- ❻ 需求 6: 判断某个单词是否包含空白符, 但除去空格
- ❼ 需求 7: 判断某个单词是否包含字母 x, 且不区分大小写
- ❽ 需求 8: 判断某个单词序列以子串开头 (或结尾), 且不区分大小写

字符串：Starts

```
struct Starts {  
    explicit Starts(const std::string& prefix) : prefix(prefix)  
    {}  
  
    bool operator()(const std::string& str) const {  
        return str.length() >= prefix.length() &&  
            str.compare(0, prefix.length(), prefix) == 0;  
    }  
  
private:  
    std::string prefix;  
};
```

字符串：Ends

```
struct Ends {  
    explicit Ends(const std::string& postfix) : postfix(postfix)  
    {}  
  
    bool operator()(const std::string& str) const {  
        return str.length() >= postfix.length() &&  
            str.compare(str.length() - postfix.length(),  
                postfix.length(), postfix) == 0;  
    }  
  
private:  
    std::string postfix;  
};
```

复用

```
template <typename T>
auto ignoring_case_starts(const T& prefix) {
    return IgnoringCase<Starts, string>(prefix);
}

template <typename T>
auto ignoring_case_ends(const T& postfix) {
    return IgnoringCase<Ends, string>(postfix);
}

exists(words.begin(), words.end(), ignoring_case_starts("abc"));
exists(words.begin(), words.end(), ignoring_case_ends("xyz"));
```

需求

- ❶ 需求 1: 判断某个单词是否包含数字
- ❷ 需求 2: 判断某个单词是否包含大写字母
- ❸ 需求 3: 判断某个单词是否包含下划线
- ❹ 需求 4: 判断某个单词是否不包含 _
- ❺ 需求 5: 判断某个单词是否包含 _，或者
- ❻ 需求 6: 判断某个单词是否包含空白符，但除去空格
- ❼ 需求 7: 判断某个单词是否包含字母 x，且不区分大小写
- ❽ 需求 8: 判断某个单词序列以子串开头 (或结尾)，且不区分大小写
- ❾ 需求 9: 判断某个单词满足某种特征，总是成功或失败

```
template <bool value, typename T>
struct Placeholder {
    bool operator()(const T&) const {
        return value;
    }
};

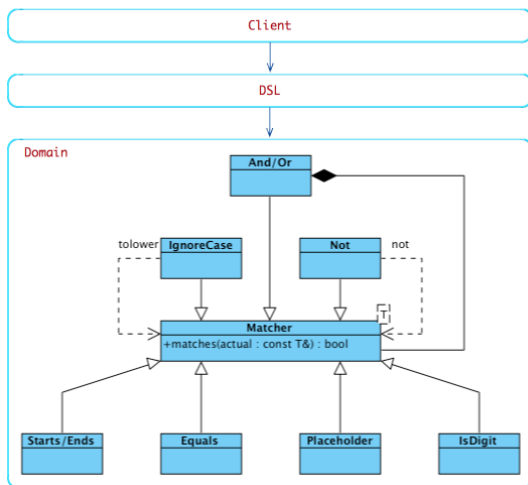
template <typename T>
auto always() {
    return Placeholder<true, T>();
}

template <typename T>
auto never() {
    return Placeholder<false, T>();
}

forall(word.begin(), word.end(), always<char>());
```

迭代 9

领域模型



函数式设计

函数式接口

函数类型

```
#include <functional>

template <typename T>
using Matcher = function<bool(const T&)>;

using CharMatcher    = Matcher<char>;
using StringMatcher  = Matcher<string>;
```


原子匹配器

```
CharMatcher is_digit() {  
    return [](auto c) {  
        return std::isdigit(c);  
    };  
}  
  
CharMatcher is_upper() {  
    return [](auto c) {  
        return std::isupper(c);  
    };  
}  
  
exists(word.begin(), word.end(), is_digit());  
exists(word.begin(), word.end(), is_upper());
```

占位符

```
template <typename T>
Matcher<T> always() {
    return [](auto) {
        return true;
    };
}

template <typename T>
Matcher<T> never() {
    return [](auto) {
        return false;
    };
}

forall(word.begin(), word.end(), always<char>());
```

逻辑相等性

```
template <typename T>
Matcher<T> equals_to(const T& expected) {
    return [=](const T& actual) {
        return expected == actual;
    };
}

exists(word.begin(), word.end(), equals_to('_'));
```

语法糖

```
Matcher<std::string> is_empty() {  
    return equals_to<std::string>("");  
}  
  
Matcher<const void*> is_nil() {  
    return equals_to<const void*>(nullptr);  
}  
  
exists(names.begin(), names.end(), is_empty());
```

组合或

```
template <typename T>
Matcher<T> is_or(Matcher<T> left, Matcher<T> right) {
    return [=](const T& actual) {
        return left(actual) || right(actual);
    };
}

exists(word.begin(), word.end(), is_or(
    equals_to('_'), equals_to('*')
));
```

组合与

```
template <typename T>
Matcher<T> is_and(Matcher<T> left, Matcher<T> right) {
    return [=](const T& actual) {
        return left(actual) && right(actual);
    };
}

exists(word.begin(), word.end(), is_and(
    is_space(), is_not(equals_to(' '))
));
```

修饰器

```
template <typename T>
Matcher<T> is_not(Matcher<T> matcher) {
    return [=](const T& actual) {
        return !matcher(actual);
    };
}

forall(word.begin(), word.end(), is_not(equals_to('_')));
```

语法糖

```
template <typename T>
Matcher<T> is_not(const T& expected) {
    return is_not<T>(equals_to(expected));
}

forall(word.begin(), word.end(), is_not('_'));
```


字符串匹配器: starts

```
StringMatcher starts(const std::string& prefix) {  
    return [=](const std::string& str) {  
        return str.length() >= prefix.length() &&  
            str.compare(0, prefix.length(), prefix) == 0;  
    };  
}  
  
exists(names.begin(), names.end(), starts("bob"));
```

字符串匹配器: ends

```
StringMatcher ends(const std::string& postfix) {  
    return [=](const std::string& str) {  
        return str.length() >= postfix.length() &&  
            str.compare(str.length() - postfix.length(),  
                postfix.length(), postfix) == 0;  
    };  
}  
  
exists(names.begin(), names.end(), ends("bob"));
```

组合器

```
template <typename T>
using MatcherFactory = function<Matcher<T>(const T&>);

template <typename T>
MatcherFactory<T> ignoring_case(MatcherFactory<T> factory) {
    return [=](const T& expected) {
        auto matcher = factory(toLower(expected));
        return [matcher](const T& actual) {
            return matcher(toLower(actual));
        };
    };
}
```

引入工厂

```
template <typename T>
Matcher<T> ignoring_case_equals(const T& expected) {
    return ignoring_case<T>(equals_to<T>)(expected);
}

StringMatcher ignoring_case_starts(const string& expected) {
    return ignoring_case<string>(starts)(expected);
}

StringMatcher ignoring_case_ends(const string& expected) {
    return ignoring_case<string>(ends)(expected);
}

exists(names.begin(), names.end(), ignoring_case_starts("liu"));
```

文献

推荐书籍

- Extreme Programming Explained: Embrace Change, 2th, Kent Beck.
- Agile Software Development: Principles, Patterns and Practices, Robert C. Martin.

联系我

- **Email:** liu.guangcong@zte.com.cn
- **Github:** <https://github.com/horance-liu>
- **简书:** <http://www.jianshu.com/users/49d1f3b7049e>

Thanks