C++ Programming Style

Guidelines, Rules, and Patterns

LTE 开发部

ZTE Corporation Copyright ©2014

This page is intentionally left blank.

目录

This page is intentionally left blank.

Do the simplest thing that could possibly work.

- Kent Beck

1 代码风格

1.1 开发环境

规则 1.1.1 为了防止代码出现可移植性问题，团队使用的操作系统、编译器类型、版本保持一致性

规则 1.1.2 团队统一使用相同的 IDE，并使用统一的代码模板，保持代码风格的一致性

系统中所有的代码看起来就好像是由单独一个值得胜任的人编写的[1]。

规则 1.1.3 团队统一配置 IDE 使用等宽字体，不允许使用宋体编程

如果团队中有人使用宋体编码，会造成团队代码对不齐、缩进混乱的情况。优秀的程序员在写第一行代码前，都会将自己的编译环境、颜色、字体等调整到最佳状态，以便在写代码的时候更友好地、更快捷地反馈所存在的问题，提高工作效率。

规则 1.1.4 团队统一配置 IDE 的文件编码格式

例如统一为 UTF-8，或 GBK, GB2312，不允许使用不同的编码格式，否则中文可能会出现乱码的现象。

规则 1.1.5 团队统一配置 IDE 的 TAB 为相同数目的空格，坚决抵制使用 TAB 对齐代码[2]

因各种编辑器解释 TAB 的长度存在不一致，如果使用 TAB 对齐代码，可能会使代码缩进混乱不堪。

[1] Agile Software Development, Principles, Patterns, and Practices, Robert C. Martin

[2] TAB 一般配置为 2 或 4 个空格，所有的编辑器都提供了类似的配置接口

1.2 代码风格

规则 1.2.1 团队应该保持一致的命名、缩进、空格、空行、断行的代码风格

业界存在多种经典的代码风格，各自都拥有独特的优势，团队应该选择并保持其中一种代码风格。

1. K&R
2. BSD/Allman
3. GNU
4. Whitesmiths

统一代码风格并非难事，团队发布统一的 IDE 代码模板，及其定制一个方便的格式化快捷键即可解决所有对齐、缩进、空格、断行等问题。

规则 1.2.2 程序实体之间有且仅有一行空行区分

函数之间的空行，能够帮组我们快速定位函数的始末的准确位置；甚至在函数内部，将逻辑相关的代码放在一起也同样具有意义，它能够帮组我们更好地理解代码块的语义。

超过一行的空行完全没有必要，部分粗心的程序员在处理这些细节时总存在着或多或少的问题，团队应该杜绝这样的情况发生。

规则 1.2.3 每个文件末尾都应该有且仅有一行空行

如果使用 Eclipse，在创建新文件时自动地在文件末添加一行空行；但诸如 Visual C++ 等情况下，则需要程序员在文件末自行手动地添加一行空行。

这样做可以最大化地实现代码可移植性，避免某些编译器因缺少空行而报告警告信息。

规则 1.2.4 每一行的代码或注释不得超过 80 列，否则将其拆分为若干行，并保持适当缩进，保证排版整洁漂亮

长表达式、长语句或多或少都存在重复，此时往往是函数提取、概念整合的最佳时机，缩短长表达式、长语句可以极大地缩短阅读代码的时间，并改善代码的可读性。

规则 1.2.5 注释符号与注释内容之间要用一个空格分割

当注释内容与注释符号之间没有空格，尤其是中文注释，导致部分编译器词法分析错误，导致编译失败[1]。

反例：

```
/*multi-line comment*/
//sigle-line comment
```

正例：

```
/* multi-line comment */
// single-line comment
```

[1]笔者曾无数次被此类低级错误问题所绊脚过。

This page is intentionally left blank.

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

2 物理设计

2.1 头文件

规则 2.1.1 自定义头文件必须具有扩展名，以区分 C++ 标准库的头文件；在团队内部，头文件、源文件扩展名类型必须保持一致性

表表2.1（第26页）列出了头文件和实现文件常见的扩展名，团队应该选择一类扩展名并保持团队内一致性。

规则 2.1.2 每一个头文件都应该具有独一无二的保护宏，并保持命名规则的一致性

命名规则包括两种风格：

1. `INCL_<PROJECT>_<MODULE>_<FILE>_H`
2. 全局唯一的随机序列码[1]

第一种命名规则问题在于：当文件名重命名或移动目录时，需要同步修改头文件保护宏；推荐使用 IDE 随机自动地生成头文件保护宏，其更加快捷、简单、安全、有效[2]。

正例：

```
#ifndef INCL_CPPUNIT_AUTO_REGISTER_SUITE_H
#define INCL_CPPUNIT_AUTO_REGISTER_SUITE_H

#endif
```

或使用 IDE 随机自动生成

[1] 例如 Eclipse 可以配置头文件的代码模板，Visual C++ 也存在类似的插件，其他 IDE 也存在类似的功能

[2] 为了简化举例，后文的头文件定义，都略去了头文件宏保护符。

```
#ifndef INCL_ADCM_LLL_3465_DCPOE_ACLDDDE_479_YTEY_H
#define INCL_ADCM_LLL_3465_DCPOE_ACLDDDE_479_YTEY_H

#endif
```

反例:

```
// 因名称太短，存在名字冲突的可能性
#ifndef ACTION_H
#define ACTION_H

#endif
```

规则 2.1.3 路径名一律使用小写、下划线或中划线风格的名称；文件名应该与程序主要实体名称相同，可以使用驼峰命名，也可以使用小写、下划线或中划线分割的名字；实现文件的名字必须和头文件保持一致；更重要的是，团队内必须保持一致的命名风格

正例:

```
#include "html-parser/core/Attribute.h"
#include "yaml_parser.h"
```

反例:

```
// 路径名 htmlParser 使用了驼峰命名风格
#include "htmlParser/core/Attribute.h"
```

规则 2.1.4 包含头文件时，必须保持路径名、文件名大小写敏感

因为在 Windows，其大小写不敏感，编译时检查失效，代码失去了可移植性，所以在包含头文件时必须保持文件名的大小写敏感。

假如存在两个物理文件名分别为 SynchronizedObject.h, yaml_parser.h 的文件。

正例:

```
#include "cppunit/core/SynchronizedObject.h"
#include "yaml_parser.h"
```

反例:

```
// 路径名、文件名大小写与真实物理路径、物理文件名称不符
#include "CppUnit/Core/SynchronizedObject.h"
#include "YAML_Parser.h"
```

规则 2.1.5 包含头文件时，路径分隔符一律使用 Unix 风格，拒绝使用 Windows 风格；即采用 / 而不是使用 \ 分割路径

正例:

```
// 使用了 Unix 风格的路径分割符
#include "cppunit/core/SynchronizedObject.h"
```

反例:

```
// 使用了 Windows 风格的路径分割符
#include "cppunit\core\SynchronizedObject.h"
```

规则 2.1.6 使用 extern "C" 时，不要包括 include 语句

正例:

```
#include "oss_common.h"

#ifdef  __cplusplus
extern "C" {
#endif

void* OSS_alloc(size_t);
void  OSS_free(void*);

#ifdef  __cplusplus
}
#endif
```

反例:

```
#ifdef  __cplusplus
extern "C" {
#endif

// 错误地将 include 放在了 extern "C" 中
#include "oss_common.h"

void* oss_alloc(size_t);
void  oss_free(void*);

#ifdef  __cplusplus
}
#endif
```

规则 2.1.7 当以 C 提供实现时，头文件中必须使用 extern "C" 声明，以便支持 C++ 的扩展

正例:

```
#include "oss_common.h"

#ifdef  __cplusplus
extern "C" {
#endif

void* oss_alloc(size_t);
void  oss_free(void*);

#ifdef  __cplusplus
}
#endif
```

反例:

```
#include "oss_common.h"

void* oss_alloc(size_t);
void  oss_free(void*);
```

建议 2.1.1 拒绝创建巨型头文件，将所有实体声明都放到头文件中

不要把所有的宏、const 常量、函数声明、类定义都要放在头文件中，而仅仅将外部依赖的实体声明放到头文件中。

实现文件也是一种信息隐藏的惯用技术，如果一些程序的实体不对外所依赖，则放在自己的实现文件中，一则可降低依赖关系，二则实现更好的信息隐藏。

巨型头文件必然造成了巨大的编译时依赖，不仅仅带来巨大的编译时开销，更重要的是这样的设计将太多的实现细节暴露给用户，导致后续版本兼容性的问题，阻碍了头文件进一步演进、修改、扩展的可能性，从而失去了软件的可扩展性。

不要认为提供一个大而全的头文件会给你的用户带来方便，用户因此而更加困扰。你应该更多地追求头文件的自满足性，而不是让你的用户还要关心头文件之间的依赖关系。

2.2 物理设计原则

物理设计是 C/C++ 语言中特有的一部分，遵循表**??**（第**??**页）中所列的设计原则，必然可以得到更加清晰、更加漂亮的头文件设计。

原则 2.2.1 *自满足原则*

所有头文件都应该自满足的。所谓头文件自满足，即头文件自身是可编译成功的。

看一个具体的示例代码，这里定义了一个 TestCase.h 头文件。

反例：

```
struct TestCase : TestLeaf, TestFixture
{
    TestCase(const std::string &name="");

private:
    OVERRIDE(void run(TestResult *result));
    OVERRIDE(std::string getName() const);

private:
    ABSTRACT(void runTest());

private:
    const std::string name;
};
```

TestCase 对父类 TestLeaf, TestFixture 都存在编译时依赖，为了满足自满足原则，其自身必须包含其所有父类的头文件。

正例：

```
#include "cppunit/core/TestLeaf.h"
#include "cppunit/core/TestFixture.h"

struct TestCase : TestLeaf, TestFixture
```

```
{
    TestCase(const std::string &name="");

private:
    OVERRIDE(void run(TestResult &result));
    OVERRIDE(std::string getName() const);

private:
    ABSTRACT(void runTest());

private:
    const std::string name;
};
```

即使 TestCase 直接持有 name 的成员变量，但没有必要包含 std::string 的头文件，因为 TestCase 覆写了其父类的 getName 成员函数，父类为了保证自满足原则，自然已经包含了 std::string 的头文件。

同样的原因，也没有必要在此前置声明 TestResult，因为父类可定已经声明过了。

规则 2.2.1 *实现文件的第一行代码必然是包含其对应的头文件*

创建对应的实现文件 TestCase.cpp，并将自身头文件的进行包含，并置在实现文件的第一行，这是验证头文件自满足的最好的办法。

正例:

```
#include "cppunit/core/TestCase.h"
#include "cppunit/core/TestResult.h"
#include "cppunit/core/Functor.h"

namespace
{
    struct TestCaseMethodFunctor : Functor
    {
        typedef void (TestCase::*Method)();

        TestCaseMethodFunctor(TestCase &target, Method method)
           : target(target), method(method)
        {}

        bool operator()() const
        {
            target.*method();
            return true;
        }

    private:
        TestCase &target;
        Method method;
    };
}

void TestCase::run(TestResult &result)
{
    result.startTest(*this);
```

```
    if (result.protect(TestCaseMethodFunctor(*this, &TestCase::setUp))
    {
        result.protect(TestCaseMethodFunctor(*this, &TestCase::runTest));
    }

    result.protect(TestCaseMethodFunctor(*this, &TestCase::tearDown));

    result.endTest(*this);
}

...
```

反例:

```
#include "cppunit/core/TestResult.h"
#include "cppunit/core/Functor.h"
// 错误：没有放在第一行，无法校验其自满足性
#include "cppunit/core/TestCase.h"

namespace
{
    struct TestCaseMethodFunctor : Functor
    {
        typedef void (TestCase::*Method)();

        TestCaseMethodFunctor(TestCase &target, Method method)
           : target(target), method(method)
        {}

        bool operator()() const
        {
            target.*method();
            return true;
        }

    private:
        TestCase &target;
        Method method;
    };
}

void TestCase::run(TestResult &result)
{
    result.startTest(*this);

    if (result.protect(TestCaseMethodFunctor(*this, &TestCase::setUp)))
    {
        result.protect(TestCaseMethodFunctor(*this, &TestCase::runTest));
    }

    result.protect(TestCaseMethodFunctor(*this, &TestCase::tearDown));

    result.endTest(*this);
}

...
```

原则 2.2.2 单一职责

这是 SRP(Single Reponsibility Priciple) 在头文件设计时的一个具体运用。头文件如果包含了其它不相关的元素，则包含该头文件的所有实现文件都将被这些不

相关的元素所污染，重编译将成为一件高概率的事件。

如示例代码，将 OutputStream, InputStream 同时定义在一个头文件中，将违背该原则。本来只需只读接口，无意中被只写接口所污染。

反例：

```
#include "base/Role.h"

DEFINE_ROLE(OutputStream)
{
    ABSTRACT(void write());
};

DEFINE_ROLE(InputStream)
{
    ABSTRACT(void read());
};

#endif
```

正例：先创建一个 OutputStream.h 文件：

```
#include "base/Role.h"

DEFINE_ROLE(OutputStream)
{
    ABSTRACT(void write());
};
```

再创建一个 InputStream.h 文件：

```
#include "base/Role.h"

DEFINE_ROLE(InputStream)
{
    ABSTRACT(void read());
};
```

原则 2.2.3 最小依赖

一个头文件只应该包含必要的实体，尤其在头文件中仅仅对实体的声明产生依赖，那么前置声明是一种有效的降低编译时依赖的技术。

反例：

```
#include <base/Role.h>
#include <cppunit/core/TestResult.h>
#include <string>

DEFINE_ROLE(Test)
{
    ABSTRACT(void run(TestResult& result));
    ABSTRACT(int countTestCases() const);
    ABSTRACT(int getChildTestCount() const);
    ABSTRACT(std::string getName() const);
};
```

如示例代码，定义了一个 xUnit 框架中的 Test 顶级接口，其对 TestResult 的依赖仅仅是一个声明依赖，并没有必要包含 TestResult.h，前置声明是解开这类编译依赖的钥匙。

值得注意的是，对标准库 std::string 的依赖，即使它仅作为返回值，但因为它实际上是一个 typedef，所以必须老实地包含其对应的头文件。事实上，如果产生了对标准库名称的依赖，基本上都需要包含对应的头文件。

另外，对 DEFINE_ROLE 宏定义的依赖则需要包含相应的头文件，以便实现该头文件的自满足。

但是，TestResult 仅作为成员函数的参数出现在头文件中，所以对 TestResult 的依赖只需前置声明即可。

正例：

```
#include <base/Role.h>
#include <string>

struct TestResult;

DEFINE_ROLE(Test)
{
    ABSTRACT(void run(TestResult& result));
    ABSTRACT(int countTestCases() const);
    ABSTRACT(int getChildTestCount() const);
    ABSTRACT(std::string getName() const);
};
```

在选择包含头文件还是前置声明时，很多程序员感到迷茫。其实规则很简单，在如下场景前置声明即可，无需包含头文件：

1. 指针
2. 引用
3. 返回值

4. 函数参数

相反地，如果编译器需要知道实体的真正内容时，则必须包含头文件，此依赖也常常称为强编译时依赖。强编译时依赖主要包括如下几种场景：

1. typedef 定义的实体[1]
2. 继承
3. 宏
4. inline
5. template
6. 引用类内部成员时
7. 执行 sizeof 运算

原则 2.2.4 最小可见性

在头文件中定义一个类时，清晰、准确的 public, protected, private 是传递设计意图的指示灯。其中 private 做为一种实现细节被隐藏起来，为适应未来不明确的变化提供便利的措施。

不要将所有的实体都 public，这无疑是一种自杀式做法。应该以一种相反的习惯性思维，尽最大可能性将所有实体 private，直到你被迫不得不这么做为止，依次放开可见性的权限。

如下例代码所示，按照 public-private, function-data 依次排列类的成员，并对具有相同特征的成员归类，将大大改善类的整体布局，给读者留下清晰的设计意图。

```
#include "trans-dsl/action/Action.h"
#include "trans-dsl/utils/EventHandlerRegistry.h"

struct SimpleAsyncAction: Action
{
    template<typename T>
    Status waitOn(const EventId eventId, T* thisPointer,
             Status (T::*handler)(const TransactionInfo&, const Event&),
             bool forever = false)
    {
        return registry.addHandler(eventId, thisPointer, handler, forever);
    }

    Status waitUntouchEvent(const EventId eventId);
```

[1]其实是弱编译时依赖，但为了避免代码重复，所以一般选择直接包含 typedef 的头文件

```
private:
    OVERRIDE(Status handleEvent(const TransactionInfo&, const Event&));
    OVERRIDE(void kill(const TransactionInfo&, const Status));

private:
    DEFAULT(void, doKill(const TransactionInfo&, const Status));

private:
    EventHandlerRegistry registry;
};
```

规则 2.2.2 所有 override 的函数必然是 private 的，以保证按接口编程的良好设计原则

反例：

```
#include "html-parser/filter/NodeFilter.h"
#include <list>

struct AndFilter : NodeFilter
{
     void add(NodeFilter*);

     // 错误：本应该 private
     OVERRIDE(bool accept(const Node&) const);

private:
     std::list<NodeFilter*> filters;
};
```

正例：

```
#include "html-parser/filter/NodeFilter.h"
#include <list>

struct AndFilter : NodeFilter
{
     void add(NodeFilter*);

private:
     OVERRIDE(bool accept(const Node&) const);

private:
     std::list<NodeFilter*> filters;
};
```

2.3 Inline

规则 2.3.1 头文件中避免定义 inline 函数，除非 profiling 报告指出此函数是性能的关键瓶颈

C++ 语言将声明和实现进行分离，程序员为此不得不在头文件和实现文件中重复地对函数进行声明。这是一件痛苦的事情，驱使部分程序员直接将函数实现为 inline。

但 inline 函数的代码作为一种不稳定的内部实现细节，被放置在头文件里，其变更所导致的大面积的重新编译是个大概率事件，为改善微乎其微的函数调用性能与其相比将得不偿失。除非有相关 profiling 的测试报告，表明这部分关键的热点代码需要被放回头文件中。

让我们就像容忍头文件中宏保护符那样，也慢慢习惯 C/C++ 天生给我们带来的这点点重复吧。

但需要注意，有两类特殊的情况，可以将实现 inline 在头文件中，因为为它们创建实现文件过于累赘和麻烦。

1. virtual 析构函数
2. 空的 virtual 函数实现
3. C++11 的 default 函数

规则 2.3.2 实现文件中鼓励 inline 函数实现

对于在编译单元内部定义的类而言，因为它的客户数量是确定的，就是它本身。另外，由于它本来就定义在源代码文件中，因此并没有增加任何“物理耦合”。所以，对于这样的类，我们大可以将其所有函数都实现为 inline 的，就像写 Java 代码那样，Once & Only Once。

以单态类的一种实现技术为例，讲解编译时依赖的解耦与匿名命名空间的使用。(首先，应该抵制单态设计的诱惑，单态其本质是面向对象技术中全局变量的替代品。滥用单态模式，犹如滥用全局变量，是一种典型的设计坏味道。只有确定在系统中唯一存在的概念，才能使用单态模式)。

实现单态，需要对系统中唯一存在的概念进行封装；但这个概念往往具有巨大的数据结构，如果将其声明在头文件中，无疑造成很大的编译时依赖。

反例：

```
#include "base/Status.h"
#include "base/BaseTypes.h"
#include "transport/ne/NetworkElement.h"
#include <vector>

struct NetworkElementRepository
{
    static NetworkElement& getInstance();

    Status add(const U16 id);
    Status release(const U16 id);
    Status modify(const U16 id);

private:
    typedef std::vector<NetworkElement> NetworkElements;
    NetworkElements elements;
};
```

受文章篇幅的所限，NetworkElement.h 未列出所有代码实现，但我们知道 NetworkElement 拥有巨大的数据结构，上述设计导致所有包含 NetworkElementRepository 的头文件都被 NetworkElement 所间接污染。

此时，其中可以将依赖置入到实现文件中，解除揭开其严重的编译时依赖。更重要的是，它更好地遵守了按接口编程的原则，改善了软件的扩展性。

正例:

```
#include "base/Status.h"
#include "base/BaseTypes.h"
#include "base/Role.h"

DEFINE_ROLE(NetworkElementRepository)
{
    static NetworkElementRepository& getInstance();

    ABSTRACT(Status add(const U16 id));
    ABSTRACT(Status release(const U16 id));
    ABSTRACT(Status modify(const U16 id));
};
```

其实现文件包含 NetworkElement.h，将对其的依赖控制在本编译单元内部。

```
#include "transport/ne/NetworkElementRepository.h"
#include "transport/ne/NetworkElement.h"
#include <vector>

namespace
{
    struct NetworkElementRepositoryImpl : NetworkElementRepository
    {
        OVERRIDE(Status add(const U16 id))
        {
            // inline implements
        }

        OVERRIDE(Status release(const U16 id))
```

```
        {
            // inline implements
        }

        OVERRIDE(Status modify(const U16 id))
        {
            // inline implements
        }

    private:
        typedef std::vector<NetworkElement> NetworkElements;
        NetworkElements elements;
    };
}

NetworkElementRepository& NetworkElementRepository::getInstance()
{
    static NetworkElementRepositoryImpl inst;
    return inst;
}
```

此处，对 NetworkElementRepositoryImpl 类的依赖是非常明确的，仅本编译单元内，所有可以直接进行 inline，从而简化了很多实现[1]。

规则 2.3.3 *实现文件中提倡使用匿名 namespace，以避免命名冲突。*

匿名 namespace 的存在常常被人遗忘，但它的确是一个利器。匿名 namespace 的存在，使得所有受限于编译单元内的实体拥有了明确的处所。

自此之后，所有 C 风格，并局限于编译单元内的 static 函数和变量；以及类似 Java 中常见的 private static 的提取函数将常常被匿名 namespace 替代。

请记住匿名命名空间也是一种重要的信息隐藏技术。

2.4 struct VS. class

建议 2.4.1 *在类定义时，建议统一使用 struct；但绝不允许混用 struct, class。*

除了名字不同之外，class 和 struct 唯一的差别是：默认可见性。这体现在定义和继承时。struct 在定义一个成员，或者继承时，如果不指明，则默认为 public，而 class 则默认为 private。

但这些都不是重点，重点在于定义接口和继承时，冗余 public 修饰符总让人不舒服。简单设计四原则告诉告诉我们，所有冗余的代码都应该被剔除。

但很多人会认为 struct 是 C 遗留问题，应该避免使用。但这不是问题，我们不应该否认在写 C++ 程序时，依然在使用着很多 C 语言遗留的特性。关键在于，我们使用的是 C 语言中能给设计带来好处的特性，何乐而不为呢？

[1] 为了简化实现和举例，使用了 STL 中的 std::vector。

正例：

```
struct SelfDescribing
{
    virtual void describeTo(Description& description) const = 0;
    virtual ~SelfDescribing() {}
};
```

反例：

```
class SelfDescribing
{
public:
    virtual void describeTo(Description& description) const = 0;
    virtual ~SelfDescribing() {}
};
```

更重要的是，我们确信“抽象”和“信息隐藏”对于软件的重要性，这促使我将 public 接口总置于类的最前面成为我们的首选，class 的特性正好与我们的期望背道而驰[1]

不管你信仰那一个流派，切忌不能混合使用 class 和 struct。在大量使用前导声明的情况下，一旦一个使用 struct 的类改为 class，所有的前置声明都需要修改。

规则 2.4.1 定义 C 风格的结构体时，摒弃 struct tag 的命名风格。

struct tag 彻底抑制了结构体前置声明的可能性，从而阻碍了编译优化的空间。

反例：

```
typedef struct tag_Cell
{
    WORD16 wCellId;
    WORD32 dwDlArfcn;
} T_Cell;

typedef struct
{
    WORD16 wCellId;
    WORD32 dwDlArfcn;
} T_Cell;
```

为了兼容 C[2]，并为结构体前置声明提供便利，如下解法是最合适的。

[1] class 的特性正好适合于将数据结构捧为神物的程序员，它们常常将数据结构置于类声明的最前面。

[2] 在 C 语言中，如果没有使用 typedef，则定义一个结构体的指针，必须显式地加上 struct 关键字：struct T_Cell *pcell，而 C++ 没有这方面的要求。

正例:

```
typedef struct T_Cell
{
    WORD16 wCellId;
    WORD32 dwDlArfcn;
} T_Cell;
```

建议 2.4.2 如果性能不是关键问题，考虑使用 PIMPL 降低编译时依赖

反例:

```
#include "JmpOnlyApiHook.h"

struct ApiHook
{
    ApiHook(const void* api, const void* stub)
      : stubHook(api, stub)
    {}

private:
    JmpOnlyApiHook stubHook;
};
```

正例:

```
struct ApiHookImpl;

struct ApiHook
{
    ApiHook(const void* api, const void* stub);
    ~ApiHook();

private:
    ApiHookImpl* This;
};
```

```
#include "ApiHook.h"
#include "JmpOnlyApiHook.h"

struct ApiHookImpl
{
   ApiHookImpl(const void* api, const void* stub)
     : stubHook(api, stub)
   {
   }

   JmpOnlyApiHook stubHook;
};

ApiHook::ApiHook( const void* api, const void* stub)
  : This(new ApiHookImpl(api, stub))
{
}
```

```
ApiHook::~ApiHook()
{
    delete This;
}
```

通过 ApiHookImpl* This 的桥接，在头文件中解除了对 JmpOnlyApiHook 的依赖，将其依赖控制在本编译单元内部。

2.5 template

建议 2.5.1 *当选择模板实现时，请最大限度地降低编译时依赖。*

当选择模板时，不得不将其实现定义在头文件中。当编译时依赖开销非常大时，编译模板将成为一种负担。设法降低编译时依赖，不仅仅为了缩短编译时间，更重要的是为了得到一个低耦合的实现。

反例：

```
#include "pub_typedef.h"
#include "pub_oss.h"
#include "oss_comm.h"
#include "pub_commdef.h"
#include "base/Assertions.h"
#include "base/Status.h"

struct OssSender
{
    OssSender(const PID& pid, const U8 commType)
      : pid(pid), commType(commType)
    {
    }

    template <typename MSG>
    Status send(const U16 eventId, const MSG& msg)
    {
        DCM_ASSERT_TRUE(OSS_SendAsynMsg(eventId, &msg, sizeof(msg),
          commType,(PID*)&pid) == OSS_SUCCESS);
        return DCM_SUCCESS;
    }

private:
    PID pid;
    U8 commType;
};
```

为了实现模板函数 send，将 OSS 的一些实现细节暴露到了头文件中，包含 OssSender.h 的所有文件将无意识地产生了对 OSS 头文件的依赖。

提取一个私有的 send 函数，并将对 OSS 的依赖移入到 OssSender.cpp 中，对 PID 依赖通过前置声明解除，最终实现如代码所示。

正例:

```
#include "base/Status.h"
#include "base/BaseTypes.h"

struct PID;

struct OssSender
{
    OssSender(const PID& pid, const U16 commType)
      : pid(pid), commType(commType)
    {
    }

    template <typename MSG>
    Status send(const U16 eventId, const MSG& msg)
    {
        return send(eventId, (const void*)&msg, sizeof(MSG));
    }

private:
    Status send(const U16 eventId, const void* msg, size_t size);

private:
    PID pid;
    U8 commType;
};
```

识别哪些与泛型相关，哪些与泛型无关的知识，并解开此类编译时依赖是 C++ 程序员的必备之技。

建议 2.5.2 适时选择 Explicit Template Instantiated，降低模板的编译时依赖。

模板的编译时依赖存在两个基本模型：包含模型，export 模型。export 模型受编译技术实现的挑战，最终被 C++11 标准放弃。

此时，似乎我们只能选择包含模型。其实，存在一种特殊的场景，是能做到降低模板编译时依赖的。

反例:

```
// quantity/Quantity.h
#include <quantity/Amount.h>

template <typename Unit>
struct Quantity
{
    Quantity(const Amount amount, const Unit& unit)
      : amountInBaseUnit(unit.toAmountInBaseUnit(amount))
    {}

    bool operator==(const Quantity& rhs) const
    {
        return amountInBaseUnit == rhs.amountInBaseUnit;
    }
```

```
    bool operator!=(const Quantity& rhs) const
    {
        return !(*this == rhs);
    }

private:
    const Amount amountInBaseUnit;
};
```

```
// quantity/Length.h
#include "quantity/Quantity.h"
#include "quantity/LengthUnit.h"

typedef Quantity<LengthUnit> Length;
```

```
// quantity/Volume.h
#include "quantity/Quantity.h"
#include "quantity/VolumeUnit.h"

typedef Quantity<VolumeUnit> Volume;
```

如上的设计，泛型类 Quantity 的实现都放在了头文件，不稳定的实现细节，例如计算 amountInBaseUnit 的算法变化等因素，将导致包含 Length 或 Volume 的所有源文件都需要重新编译。

更重要的是，因为 LengthUnit, VolumeUnit 头文件的包含，如果因需求变化需要增加支持的单位，将间接导致了包含 Length 或 Volume 的所有源文件也需要重新编译。

如何控制和隔离 Quantity, LengthUnit, VolumeUnit 变化的蔓延，而避免大部分的客户代码重新编译，从而与客户彻底解偶呢？可以通过显式模板实例化将模板实现从头文件中剥离出去，从而避免了不必要的依赖。

```
// quantity/Quantity.h
#include <quantity/Amount.h>

template <typename Unit>
struct Quantity
{
    Quantity(const Amount amount, const Unit& unit);

    bool operator==(const Quantity& rhs) const;
    bool operator!=(const Quantity& rhs) const;

private:
    const Amount amountInBaseUnit;
};
```

```
// quantity/Quantity.tcc
#include <quantity/Quantity.h>

template <typename Unit>
Quantity<Unit>::Quantity(const Amount amount, const Unit& unit)
  : amountInBaseUnit(unit.toAmountInBaseUnit(amount))
{}

template <typename Unit>
bool Quantity<Unit>::operator==(const Quantity& rhs) const
{
    return amountInBaseUnit == rhs.amountInBaseUnit;
}

template <typename Unit>
bool Quantity<Unit>::operator!=(const Quantity& rhs) const
{
    return !(*this == rhs);
}
```

```
// quantity/Length.h
#include "quantity/Quantity.h"

struct LengthUnit;

struct Length : Quantity<LengthUnit> {};
```

```
// quantity/Length.cpp
#include "quantity/Quantity.tcc"
#include "quantity/LengthUnit.h"

template struct Quantity<LengthUnit>;
```

```
// quantity/Volume.h
#include "quantity/Quantity.h"

struct VolumeUnit;
struct Volume : Quantity<VolumeUnit> {};
```

```
// quantity/Volume.h
#include "quantity/Quantity.tcc"
#include "quantity/VolumeUnit.h"

template struct Quantity<VolumeUnit>;
```

Length.h 仅仅对 Quantity.h 产生依赖; 特殊地，Length.cpp 没有产生对 Length.h 的依赖，相反对 Quantity.tcc 产生了依赖。

另外，Length.h 对 LengthUnit 的依赖关系也简化为声明依赖，而对其真正的编译时依赖，也控制在模板实例化的时刻，即在 Length.cpp 内部。

LenghtUnit, VolumeUnit 的变化，及其 Quantity.tcc 实现细节的变化，被完全地控制在 Length.cpp, Volume.cpp 内部。

建议 2.5.3 子类化优于 typedef

正例：

```
#include "quantity/Quantity.h"
struct LengthUnit;
struct Length : Quantity<LengthUnit> {};
```

当仅仅对 amountInBaseUnit 声明产生依赖时，前置声明即可，无需包含整个 Length 的头文件。另外，如果 Quantity 存在 virtual 函数时，Length 还有进一步扩展 Quantity 的可能性，从而使设计提供了更大的灵活性。

如果使用 typedef，如果存在对 Length 的依赖，即使是名字的声明依赖，除了包含头文件之外，别无选择。

反例：

```
#include "quantity/Quantity.h"
struct LengthUnit;
typedef Quantity<LengthUnit> Length;
```

文件类型	支持的扩展名
头文件	.h, .hpp, .hxx, .hh, h++, .tcc
源文件	.c, .C, .cpp, .cxx, .cc, .c++

表 2.1 扩展名

原则	基本含义
自满足原则	头文件本身是可以编译通过的
单一职责原则	头文件包含的实体的职责是单一的
最小依赖原则	绝不包含不必要的头文件
最小可见性原则	尽量封装隐藏类的成员

表 2.2 物理设计四原则

Controlling complexity is the essence of computer programming.

- Brian Kernighan

3 不可变性

我们无时无刻都在面临着新的变化，优秀的程序员擅于控制变化，尽量让对象变得不可变 (immutable)，以便降低问题的复杂度。

3.1 const

规则 3.1.1 使用 const 替代 #define 宏常量定义

使用 const 替代宏常量理由在经典的《Effective C++》著作中进行了详尽的描述，但往往受 C 语言根深蒂固的习惯常常被人遗忘。

规则 3.1.2 当成员函数具有查询语义时应声明为 const 成员函数

此处具有查询语义的函数，泛指所有对对象状态未产生副作用的函数。const 成员函数往往都以 get, is, has, should 开头。

以 Ruby 中对 RubyIdentity 的实现为例[1]。

反例:

```
#include "base/BaseTypes.h"

struct Rectangle
{
    Rectangle(const U8 width, const U8 height);

    U16 getArea();
        U16 getPerimeter();

private:
    const U8 width;
        const U8 height;
};
```

此例中未对那些具有查询语义的函数声明为 const。const 的存在不仅仅是为了提高编译时的安全性检查，更重要的是 const 建立了用户和类之间的契约关系，并传达了程序员良好设计的心声。

[1] RubyIdentity 在内存中的映虽然仅仅是一个整数而已，依然提倡对它存在的逻辑进行封装，系统因为这些小类的存在变得更加具有可读性和可复用性。

当对象以 pass-by-const-reference 传递时，用户只能调用其 const 成员函数；此外，const 建立的契约，使 Replace Temp with Query 成为可能。

正例：

```
#include "base/BaseTypes.h"

struct Rectangle
{
    Rectangle(const U8 width, const U8 height);

    U16 getArea() const;
        U16 getPerimeter() const;

private:
    const U8 width;
        const U8 height;
};
```

规则 3.1.3 const 成员函数不应该修改系统的状态

编译器只能保证 const 成员函数修改成员变量时提示错误，而未对修改全局变量，类的静态变量的情况进行检查，应杜绝此类情况的发生。

规则 3.1.4 以 pass-by-reference-to-const 替代 pass-by-value，以改善性能，并避免切割问题。

虽然我们不提倡进行过早的优化，但也绝不提倡不成熟的劣化。pass-by-reference-to-const 就是此句名言最好的一个例子。

建议 3.1.1 当传递内置类型，迭代器及函数对象时，则可以 pass-by-value。如果使用 const 修饰，往往可以改善 API 的可读性

其中，Status, ActionId 分别是 U32, U16 的 typedef，但在设计 onDone 原型时，const 的修饰不仅仅为了改善其安全性，更重要的是与 TransactionInfo 的 reference-to-const 形成对称性[1]，改善了 API 的美感。

反例：

[1]关于对称性，请参考 Kent Beck 的著作《实现模式》

```
#include "base/Status.h"
#include "trans-dsl/concept/ActionId.h"

struct TransactionInfo;

struct TransactionListener
{
    virtual Status onDone(const TransactionInfo&, const ActionId&, const
      Status&)
    {}
};
```

正例:

```
#include "base/Status.h"
#include "trans-dsl/concept/ActionId.h"

struct TransactionInfo;

struct TransactionListener
{
    virtual Status onDone(const TransactionInfo&, const ActionId, const
      Status) {}
};
```

规则 3.1.5 *当返回的是一个新创建的对象时，必须返回其值类型；更有甚者，返回* const *的值类型将改善编译时的安全性*

反例:

```
#include "base/BaseTypes.h"

struct GlobalIdentity
{
    explicit GlobalIdentity(const U16 id);

    bool isValid() const;

    bool operator==(const Gid& rhs) const;
    bool operator!=(const Gid& rhs) const;

    static Gid& selfGid();
    static Gid& valueOf(U16 value);
};
```

静态工厂方法 valueOf, selfGid 返回的是一个新创建的实例，所以返回值必须设计为值类型，提供 const 的修饰能够加强编译时的安全性检查。

正例:

```
#include "base/BaseTypes.h"

struct GlobalIdentity
{
    explicit GlobalIdentity(const U16 id);

    bool isValid() const;

    bool operator==(const Gid& rhs) const;
    bool operator!=(const Gid& rhs) const;

    static const Gid selfGid();
    static const Gid valueOf(U16 value);
};
```

规则 3.1.6 不能返回局部对象的引用或指针。

返回值类型还是引用类型常常会困扰部分 C++ 程序员。其实规则非常简单，返回值对象当且仅当需要创建新对象的时候，此时如果企图返回引用或指针，将导致运行时异常。

3.2 不可变类

建议 3.2.1 鼓励设计 Immutable Class 表达 Value Object 的语义

不可变类的每一个实例在创建的时候就包含了所有必要的信息，其对象在整个生命周期中都不会发生变化。不可变类具有如下几方面的优势:

1. 相对于可变对象的复杂的状态空间，不可变类仅有一个状态，容易设计、实现和控制
2. 不可变类本质上是线程安全的，它们不需要同步
3. 不可变类可被自由地共享，重用对象是一种良好设计的习惯

设计不可变类，需遵循如下规则:

1. 不能提供修改对象状态的方法
2. 保证类不能被扩展，切忌提供 virtual destructor
3. 所有域都是 const
4. 所有域都是 private
5. 绝不允许重新赋值

不可变类唯一的缺点就是，对于每一个不同的值都需要一个单独的对象。当需要创建大量此类对象时，性能可能成为瓶颈。当性能成为关键瓶颈的时候，为不可变类设计可变配套类是一种典型的设计方式。

例如 Java 的 JDK 中，String 被设计为不可变类，但在诸如字符串需要大量的替换、追加等场景下，性能可能成为关键瓶颈；JDK 提供了 StringBuilder 的可变配套类来解决此类问题。

上例讲解的 Rectangle 类也是一种典型的不可变类的设计。

规则 3.2.1 尽量使类的可变性最小化

坚决抵制为每一个 get 函数都写一个 set 函数，除非存在一个很好的理由。当写一堆无聊的 get/set 就自居精通面向对象的精髓是幼稚的。

构造函数的职责就是来完成对象的初始化，并建立起所有的约束关系。除非有很好的理由，不要在构造函数之外提供其他公开的初始化方法。

This page is intentionally left blank.

Write programs for people first,
computers second.
- Steve McConnell

4 命名

良好的命名将改善代码的表现力，与其说命名是一门技术，不如说它是一门艺术。必须坚持、并慎重地给程序中的每一个实体取好名字，让其名副其实，代码的可读性将得到极大的改善。

4.1 Baby Names

规则 4.1.1 *遵守统一的命名规范*

一些程序设计语言，从诞生的时刻就有着自己的文化背景，整个社区有着统一的命名规则。但 C++ 在社区中存在众多的命名风格，有的与时俱进，也非常人性化；有的则已经与现代软件工程格格不入[1]。

下文罗列了一些 C++ 典型的命名规范供参考。每一种命名规范都有自身自己的优缺点，团队应该根据自身的实际情况选择适合自己的命名规范。最重要的是，团队应遵循统一的命名规范，不应该出现两种或两种以上的命名风格。

我们推荐团队使用下文中任意一种命名风格，但如果由于历史遗留原因导致革新非常困难，而无法采用新的命名规范，团队依然要保持统一的命名规范，避免出现多种命名风格。

还有一种命名风格与上一种风格类同，只是成员函数/函数都以大写开头的驼峰命名。

第三种命名风格，主要体现在标准库或 Boost 准标准库社区，其规则非常简单，下划线分割的全小写，或下划线分割的全大写。

规则 4.1.2 *绝不使用汉语拼音命名*

规则 4.1.3 *类名应该是名词或名词短语；接口可以是形容词；方法名应该是动词或动词短语*

[1] 例如社区中依然存在一批忠实的匈牙利的守护者，他们认为一个名称没有前缀和后缀标识它们的类型，他们几乎都看不懂代码了。

Identifier	Examples
Namespace	std, dcm, mockcpp, testing
Class/Struct/Union	Timer, FutureTask, LinkedHashMap, HttpServlet
Method	remove, ensureCapacity, getCrc
Constant/Macro/Enum	IDLE, ACTIVE, MAX_LINK_NUM
Local Variable	i, xref, houseNumber
Type Parameter	T, E, K, V, X, T1, T2

表 4.1 驼峰命名规范 1

Identifier	Examples
Namespace	std, dcm, mockcpp, testing
Class/Struct/Union	Timer, FutureTask, LinkedHashMap, HttpServlet
Method	Remove, EnsureCapacity, GetCrc
Constant/Macro/Enum	IDLE, ACTIVE, MAX_LINK_NUM
Local Variable	i, xref, houseNumber
Type Parameter	T, E, K, V, X, T1, T2

表 4.2 驼峰命名规范 2

Identifier	Examples
Namespace	boost, details, mpl
Class/Struct/Union	any, is_enum, shared_ptr
Method	any_cast, type_of
Constant/Macro/Enum	IDLE, ACTIVE, MAX_LINK_NUM
Local Variable	i, xref, house_number
Type Parameter	T, E, K, V, X, T1, T2

表 4.3 标准库或 boost 命名规范

类别	举例
正确的类名	AddressParser, EventRegistry
错误的类名	ParseAddress, RegisterEvent

表 4.4 类名

接口可以是形容词，最为常见的就是定义能力接口，即以 -able 结尾。

正例:

```
#include "base/Keywords.h"

DEFINE_ROLE(Runnable)
{
    ABSTRACT(void run());
};
```

规则 4.1.4 *如果函数返回值为* bool，*加上* is, has, can, should, need *将会增强语意*

反例:

```
bool readPassword = true;
```

至少存在两种解释,

1. We need to read the password
2. The password has already been read

正例:

```
bool needPassword = true;
bool userIsAuthenticated = true;
```

规则 4.1.5 *丰富你的单词库，在面对具体问题时你具有更多的* Colorfull Words

如表4.5（第42页）所示，同一概念是有很多种表达方式的。

规则 4.1.6 *名字在明确意图的前提下越短越好*

反例:

```
ControllerForEfficientHandlingOfStrings
ControllerForEfficientStorageOfStrings
```

因为这两个类名实在是太相似了，一时间很难区分。当别人打算复用你的代码时，必然承受着巨大的记忆包袱。

另外，实体名称的长度应该于作用域的大小成正比。如果是一个实体暴露在全局命名空间，则需要适当增加名字长度，防止名字冲突[1]。相反地，在一个很小的函数之内可见的局部变量，尤其在一个短小的 for 循环作用域内，完全没有必要取一个很长的名字。

反例:

```
void MutilEventListener::onEventDone()
{
    for(int index=0; index!=listeners.size(); index++)
    {
        listeners[index].onEventDone();
    }
}
```

优秀的程序员在命名的长短的取舍总是游刃有余，但从来没有降低过代码的意图。

正例:

```
void MutilEventListener::onEventDown()
{
    for(int i=0; i!=listeners.size(); i++)
    {
        listeners[i].onEventDone();
    }
}
```

规则 4.1.7 *程序中每个实体都应该有一个意图明确 (Intention-Revealing) 的名称*

```
vector<vector<int> > getThem()
{
    vector<vector<int> > list1;

    for(auto x : theList)
    {
        if(x[0] == 4)
        {
            list1.add(x);
        }
    }

    return list1;
}
```

[1]应该使用命名空间，避免增加模块前缀信息。

1. vector<vector<int> > 的语法令人抓狂
2. getThem 让人看不清楚它的本意
3. theList 到底是什么东西?
4. 0 下标意味着什么? 4 又意味着什么?
5. list1 就是为了编译通过吗?

换一个好名字之后，并对数据结构进行了简单的封装。

```
#include <vector>

struct Cell
{
    bool isFlagged() const;

private:
    std::vector<int> states;
};

using Cells = std::vector<Cell>;
```

重构之后的效果，抽象和可读性得到了改善[1]。

```
Cells getFlaggedCells() const
{
    Cells flaggedCells;

    for(auto &cell : gameBoard)
    {
        if(cell->isFlagged())
        {
            flaggedCells.push_back(*cell);
        }
    }

    return flaggedCells;
}
```

规则 4.1.8 避免在名称中携带数据结构的信息

别用 accountList，accountArray 指定一组账号，当包含 Account 的容器不在是一个 List 或 Array 的时候，就会引发错误的判断。所以，用 accountGroup，bunchOfAccounts，甚至直接使用 accounts，情况都会更好一些。

规则 4.1.9 Noise Words are Redundant，消除噪声后将得到一个更加精准的名字

如表4.6（第42页）所示，消除冗余的噪声，将得到一个更加直观、更漂亮的名字。

[1]此处为了方便举例，使用了 C++11 的 for-each, using 别名定义的语法，简化迭代器的操作。

规则 4.1.10 使用 Domain 领域内的名称，将更直白地表明你的设计，增强领域内成员的沟通

1. Factory, Visitor, Repository
2. valueOf, of, getInstance, newInstance, newType
3. AppendAble, Closeable, Runnable, Readable, Invokable

当使用 Visitor，你在使用访问者模式；当使用 valueOf, of，你在使用静态工厂方法。领域内的命名风格，让领域内的成员更加快捷地理解你的设计意图。

4.2 匈牙利命名

建议 4.2.1 摒弃匈牙利命名

匈牙利命名曾风靡一时。但是，现代编程语言具有更丰富的类型系统；人们更趋于使用更小的类，更短的函数，让每一个变量定义都在可控的视野范围之内；此外，IDE 变得越来智能和强大，匈牙利命名反而变成了一种噪声和肉刺。

建议 4.2.2 摒弃给成员变量加前缀，或后缀

为类的成员变量增加 m_ 前缀同样没有必要。与其在在犹豫加前缀还是不加前缀，不如花费更多的时间将类分解；当类足够小，职责足够单一，使用现代 IDE 的着色功能，成员变量和普通变量一眼便能识别开来[1]。

建议 4.2.3 摒弃常量的前缀

常量前增加 k_ 前缀同样没有必要，如果熟悉了大写、下划线隔开的命名风格，k_ 前缀完全属于多余。

建议 4.2.4 摒弃接口和类的前缀

在接口前增加 I_，在类前增加 C_，同样完全没有必要，它只能对重构带来阻力，百害而无一利。

[1]Eclipse 默认使用蓝色与普通变量区别开来，其他 IDE 也提供类似的功能。

```
#include "base/Keywords.h"
#include "base/Status.h"

struct TransactionInfo;

DEFINE_ROLE(IAction)
{
    ABSTRACT(Status exec(const TransactionInfo&));
};
```

Action 前面的 I 就是一句废话。如果非得在接口与实现中选择，宁愿选择实现中增加 Impl 后缀。

4.3 注释

规则 4.3.1 注释不如花费更多时间给实体取一个好名字，让其名副其实

反例：

```
int time; // elapsed time in days
```

正例：

```
int elapsedTimeInDays;
```

注释应成为一种羞耻的活动，当需要添加注释的时候，往往是改善设计的最好时机。

规则 4.3.2 消除所有没有必要的注释

没有携带任何信息量的注释都是没有必要的。如下列的注释，维护它简直就是一种巨大的包袱，有时候它的存在就像一个笑话。它就像在挑战读者的智商，谁都知道它是一个构造函数。

反例：

```
/**
 * Constructor
 */
InvokedAtMost(const unsigned int times);
```

```
/**
 * @param Invocation
 * @return bool
 */
bool matches(const Invocation& inv) const;
```

规则 4.3.3 消除所有误导性、过时的、与设计实现不匹配的注释

如果注释和代码已经失去匹配，意图甚至是相反的。这样的注释如果继续存在，贻害的肯定不止一个人。“假如代码和注释不一致，那么很可能两者都是错误的”，Norm Schryer 形象地描述了不一致的注释给代码维护者带来的困扰。

规则 4.3.4 消除所有日志型、归属、签名的注释

修正一个 bug 时，都需要在函数头的注释表中记录被次修改的内容。放弃这种“好”习惯吧，请将这些详细的信息提交给源代码控制系统，那里是此类注释最好的归所。

规则 4.3.5 消除所有//end if, //end while, //end for, // end try 的注释

因为逻辑太过于冗长、复杂，又为了增强可读性，往往在花括号后面标记//end if, //end while, //end for, // end try，这往往是简化表达式、提取函数的最佳时机。

规则 4.3.6 已经被注释掉的代码，应该立即删除

不要珍惜这些冗余的代码，害怕丢掉这些代码而买不到后悔药。你要相信，从源代码控制系统中追回这份被删除的代码易如反掌。

规则 4.3.7 在需要注释的时候，一定要加注释

因公司版权的保护，需要增加版权法律信息注释[1]。幸运的是，这类问题可以通过代码模板轻松解决，不会成为造成程序员的任何负担。

在已经尽全力也无法用代码表述清楚时，增加必要的注释会使代码更加容易被别人理解和维护。在这些场景下，注释将成为我们最后一根救命稻草。

1. 代码无法明确的意图也需要增加注释

[1]发布开源代码时，也常常需要增加必要的 GNU, BSD 等公共许可证

2. 如果在代码中存在特殊的陷阱、实现手法等情况，此时注释变得尤为宝贵

例如，当操作复杂的位运算时，提供比特位的内存映像的注解，将有利于加快读者阅读代码的速度。

正例：

```
// Fast version of "hash = (65599 * hash) + c"
hash = (hash << 6) + (hash << 16) - hash + c;
```

当定义难以理解的正则表达式时，提供更直观的格式说明，同样可以改善对代码的理解。

正例：

```
// kk::mm::ss, MM dd, yyyy
std::string timePattern = "\\d{2}:\\d{2}:\\d{2}, \\d{2} \\d{2}, \\d{4}";
```

Word	Alternatives
send	deliver, dispatch, announce, distribute, route
find	dsearch, extract, locate, recover
start	launch, create, begin, open
make	create, set up, build, generate, compose, add, new

表 4.5 Colorfull Words

Short Name	Redundant Names
Name	StrName, NameString
Customer	CustmerObject, CustmerInfo
accouts	accountList, accountArray
accout	accountData, accountInfo
money	moneyAmount
message	theMessage

表 4.6 消除冗余的噪声

Do the simplest thing that could possibly work.

- Kent Beck

5 简化逻辑

5.1 简化控制逻辑

规则 5.1.1 YODA Notation 已经是过去时

```
if (length <= MAX_STREAM_LEN)
if (MAX_STREAM_LEN >= length)
```

前者更具有表达力，再看一个例子，

```
while (bytesReceived < bytesExpected)
while (bytesExpected > bytesReceived)
```

同样地，前者更具有表达力。因为前者更加符合人类的思维，或这说更符合英语的表达习惯，”if you are at least 18 years old.” 明显比”if 18 years is less than or equal to your age” 更加符合英语表达习惯。是的，英语更加习惯将稳定的一侧放在右边 (right-hand side)。

但是，在 C 或者 C++ 语言中，如果 == 被误用为 =，则可能发生副作用的危险，而且编译时并不能做到安全的检查，从而产生了严重的 bug。

```
// 缺少等号，造成严重的运行时错误
if (ptr = NULL)
```

幸运的是，现代编译器对此类误用通常报告警告信息；此外如果保持 TDD 开发节奏，小步前进，此类低级错误很难逃出我们的法眼。

规则 5.1.2 在简单逻辑的情况下，鼓励使用?: 三元表达式；但是，当表达式变得非常复杂时，if-else 可能是更好的选择

反例：

```
if (hour >= 12)
{
    time += "pm";
}
else
{
    time += "am";
}
```

正例:

```
time_str += (hour >= 12) ? "pm" : "am";
```

但很多程序员往往因为习惯了 `if-else` 而忽视了这种简洁的表达式。但是，当表达式变得非常复杂时，`if-else` 可能是更好的选择。

反例:

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 <<
  -exponent);
```

正例:

```
if (exponent >= 0)
{
    return mantissa * (1 << exponent);
}
else
{
    return mantissa / (1 << -exponent);
}
```

规则 5.1.3 *当逻辑表达式已经具有* true *或* false *语义时，则无需显示地加上* true *或* false

反例:

```
bool exist = erabs.contains(ErabId(2)) ? true : false;
if(exist == true)
{
    ...
}
```

正例:

```
if(erabs.contains(ErabId(2))
{
    ...
}
```

规则 5.1.4 避免函数嵌套过深，擅用卫语句从函数中提前返回，或从嵌套的语句中提前返回；但绝对不滥用 break, continue

嵌套过深的函数，将迫使读者跟踪运行时 stack，同样给读者增加过重的记忆包袱。

反例：

```
if (userResult == SUCCESS)
{
    if (permissionResult != SUCCESS)
    {
        reply.writeError(permissionResult);
        return;
    }
    reply.writeError("");
}
else
{
    reply.writeError(usrResult);
}
```

反例：

```
if (userResult != SUCCESS)
{
    reply.writeError(usrResult);
    return;
}

if (permissionResult != SUCCESS)
{
    reply.writeError(permissionResult);
    return;
}

reply.writeError("");
```

虽然在代码阅读过程中 break, continue, return 常常打断了人的思维。但在函数短小，意图明确的情况下，break, continue, return 相反能给人一种“跳过此项”的特殊意图，代码嵌套逻辑得到了改善。

规则 5.1.5 擅用解释型变量，或提供意图明确的、内联的提取函数

解释型变量，或提取函数的重构手法有利于改善代码的可读性，往往后者更具有表达力。

反例：

```
if (line.split(":")[0].trim() == "root")
```

正例：

```
String userName = line.split(":")[0].trim();
if (userName == "root")
{
    ...
}
```

split 函数将字符串按照指定的正则表达式进行拆分，并提供一个良好的意图变量，将大大改善逻辑的清晰度。

规则 5.1.6 缩小变量的作用域

规则 5.1.7 将局部变量的作用域最小化

这条规则可能对从 C 转变过来的 C++ 程序员更有指导意义，但改变这样的陋习可能需要一些适应的时间。例如局部于 for 的循环变量，当它的使命完成之后，应立即销毁它，避免这个变量被再次使用的风险。

```
map<string, string> addressBook;
for (auto &entry : addressBook)
{
    std::cout << entry->first << "," << entry->second << std::endl;
}
```

同样的规则也使用与其他常见的情况，例如 if 语句，简化空指针的判断，简洁有效。

```
if (PaymentInfo* info = database.readPaymentInfo())
{
    std::cout << "User paid: " << info->amount() << std::endl;
}
```

但也有人对于 if 语句这样的用法持反对态度，因为他们害怕这样做更容易出错。但在最坏的情况下，即使误写成了 ==，也只是编译时错误。

I'm not a great programmer;
I'm just a good programmer
with great habits.

- Kent Beck

6 类设计

6.1 构造与析构

规则 6.1.1 抵制设计和实现上帝类

上帝类是一种典型的代码坏味道，程序员几乎都曾经历过被巨类所困扰的经历。20-30 多个成员变量在成员函数中穿梭，犹如像全局变量一样难理解和控制。此外，修改这个巨类需要高超的技能，因为你根本不确定代码的修改是否会影响其他的东西。

消除这样的巨类，关键是识别出类中的主要职责，按照 SRP 原则来分解巨类，将类拆分成为一个个细小粒度的、高度可复用的、可组合的小类[1]。

规则 6.1.2 明确 C++ 的初始化规则

初始化永远是程序员心中的痛，抛开一些不重要的细节，其实 C++ 语言的初始化规则非常简单。C++ 使用构造函数完成类初始化，而构造函数使用初始化列表完成初始化的工作。初始化列表首先调用父类的构造函数，然后按照类中定义的的成员变量的顺序依次进行初始化。

没有显式地调用父类构造函数，编译器则自动调用父类的默认构造函数；如果对应的父类没有默认构造函数，则需显式地调用特定的带参构造函数，否则编译失败。

依次类推，如果没有显式地调用成员变量的构造函数，则编译器自动地调用成员变量的默认构造函数；如果对应的成员变量没有默认构造函数，则需显式地调用特定的带参构造函数，否则编译失败。

一般地，如果已经明确需要调用默认构造函数，为了消除冗余代码，则无需显式地进行调用。

特殊地，const 数据成员，引用数据成员必须在初始化列表中进行初始化。

[1] 详细内容可参考 Michael C. Feathers 所著的《Working Effectively with Legacy Code》

规则 6.1.3 *在初始化列表中显式地初始化所有内置数据类型的成员变量，C++ 语言并没有保证其初始化*

基本数据类型包括整形，浮点型，指针，引用等类型。当设计一个含有基本数据类型的成员变量时，必须定义构造函数完成所有成员变量的初始化。

TransData 必须实现对基本数据类型的成员变量的初始化，否则行为未定义。至于泛型类型 T 则要求必须存在默认构造函数，否则编译失败。

```
template<typename T>
struct TransData
{
   TransData();
   ~TransData();

   void confirm()
   void revert()

private:
   T values[2];
   unsigned char valid :1;
   unsigned char memoed :1;
   unsigned char unstable :1;
   unsigned char current :1;
   unsigned char :4;
};

template<typename T>
TransData<T>::TransData()
  : valid(0), memoed(0), unstable(0), current(0)
{
}
```

规则 6.1.4 *在初始化列表中，而非在构造函数体内进行初始化*

如果类类型的成员变量没有在初始化列表中进行初始化，事实上编译器调用的是默认构造函数 (如果没有默认构造函数，需要显式地调用带参构造函数；否则编译错误)。

这里需要明确地区分初始化与赋值的语义。如果将类类型的初始化放在构造函数体内，则编译器首先在初始化列表中调用默认构造函数，然后再在其构造函数体内调用 operator= 重新赋值。这样的行为往往是低效的，不是我们所期望的初始化行为。

规则 6.1.5 *初始化列表顺序与成员变量定义的顺序保持一致*

因为编译器的初始化是按照类中成员变量的声明顺序依次进行初始化，如果违背这个规则，可能出现初始化顺序依赖的问题。

规则 6.1.6 除非确认编译器能够正确地调用对应父类或成员变量的默认构造函数，并符合预期行为；否则需要显式地调用其带参构造函数完成初始化

相反地，如果已经确认编译器能够正确地调用对应父类或成员变量的默认构造函数，并符合预期行为，无需再显示地调用其默认构造函数，否则将产生冗余的代码。

规则 6.1.7 使用 RAII 完成资源的安全管理

RAII(Resource acquisition is initialization) 是一种有用的技术，可实现了资源的安全管理功能。

```
#include "thread/mutex.h"

struct Lock
{
    Lock(Mutex &mutex) : mutex(mutex)
    {
        mutex.lock();
    }

    ~Lock()
    {
        mutex.unlock();
    }

private:
    Mutex& mutex;
};

#define SYNCHRONIZED(mutex) Lock mutex##_lock(mutex)
```

RAII 利用了局部对象的自动销毁的机制，实现资源的自动释放的功能。

```
template<typename T>
void BlockQueue<T>::push(const T& item)
{
    SYNCHRONIZED(mutex);

    // others
}
```

规则 6.1.8 在特定场景下延迟初始化对象

在延迟初始化之前，必须预先占有对象所需要的原始内存空间。但被延迟的对象类型如果在如下两种场景，则无法预先占有原始内存。

1. union 内部的放置了非 POD 类型[1]
2. 被延迟初始化的对象无默认构造函数可调用

一般地，可以通过定义默认构造函数预先占用好对象内存的空间，当延迟初始化时再重新赋予新的值。但这完全不符合延迟初始化的概念，延迟初始化目标就是为了完全避免这次无谓的默认构造的开销。

可以引入类 Placement，完成内存的预先占用，当延迟初始化发生时，在此 Placement 的原始内存上调用 placement new 完成对象的延迟构造；并在此原始内存上提供了一层直观的、人性化的操作接口，从而可以很方便地操作被修饰的类型 T 对象。

```
#include <string.h>
#include <new>

template <typename T>
struct Placement
{
    void* alloc()
    {
        ::memset(u.buff, 0x00, sizeof(T));
        return u.buff;
    }

    void dealloc()
    {
        getObject()->~T();
    }

    T* operator->() const
    {
        return (T*)u.buff;
    }

    T& operator*() const
    {
        return *(T*)u.buff;
    }

    T* getObject() const
    {
        return (T*)u.buff;
    }

private:
    union
    {
        char   c;
        short  s;
        int    i;
        long   l;
        float  f;
        double d;
        void*  p;

        char buff[sizeof(T)];
    }u;
};
```

[1]C++11 已经增强了 union 的语义，union 内可放置非 POD 类型的成员变量了。

我们分别看两个例子。第一个例子，如果存在两个非 POD 类型，它们被放在 union 中，只有在延迟初始化时才能决定其真正的类型的构造。

```
#include <base/BaseTypes.h>

struct Account;

struct AccountFactory
{
    static Account* create(const U8 type);
};
```

```
#include <money/LocalAccount.h>
#include <money/RemoteAccount.h>
#include <new>

namespace
{
    union AccountMemery
    {
        Placement<LocalAccount>  local;
        Placement<RemoteAccount> remote;
    } m;
}

Account* AccountFactory::create(const U8 type)
{
    switch(type)
    {
    case LOCAL_ACCOUNT:
        return new (m.local.alloc()) LocalAccount;
    case LOCAL_ACCOUNT:
        return new (m.remote.alloc()) RemoteAccount;
    default: break;
    }

    return 0;
}
```

如果一个类包含一个类类型的数组，除非提供数组元素类型的默认构造函数，否则编译失败。使用 Placement 预先占用数组所需的内存，再在合适的时机再进行延迟初始化，则可以避免了数组元素的默认构造，提升了效率。

```
#include <base/Placement.h>
#include <radio/Sensor.h>

struct SensorRepository
{
    SensorRepository() : num(0)
    {}

    ~SensorRepository()
    {
        for(U16 i=0; i<num; i++)
        {
            sensors[i].~Sensor();
        }
    }
```

```
    void addSensor(const U8 id)
    {
        if(num < MAX_NUM_SENSOR)
        {
            new (sensors[num++].alloc()) Sensor(id);
        }
    }

private:
    enum { MAX_NUM_SENSOR = 1024 };

    U16 num;
    Placement<Sensor> sensors[MAX_NUM_SENSOR];
};
```

有了 Placement，此时 Sensor 就没有必要定义默认构造函数，在延迟初始化的时调用真正的带参构造函数。

规则 6.1.9 *绝不 public 内部的成员变量*

如果公开了内部成员变量，则违背了信息隐藏机制，这是一种典型的坏味道。相反地，当遇到需要操作遗留系统中 C 语言的结构体，需要对其进行封装，并提供相应的成员函数，让其更加符合面向对象的思维。

```
template <typename From, typename To>
struct StructWrapper : protected From
{
    static const To& by(const From& from)
    {
        return (const To&)from;
    }

    static To& by(From& from)
    {
        return (To&)from;
    }
};

#define STRUCT_WRAPPER(To, From) strict To : StructWrapper<From, To>
```

使用 StructWrapper 提供的静态工厂方法 by，可方便地实现 C 结构体转变为封装了的 C++ 类。

规则 6.1.10 *遇到多个构造器参数时考虑用构建器*

规则 6.1.11 *考虑使用静态工厂方法代替构造函数*

static factory method 具有很多优势，

1. 具有比构造函数更丰富的名称
2. 不必每次调用时都创建一个对象，实现对象复用，提升性能
3. 返回类型为接口类型，遵循按接口编程的良好设计原则

以实现对美元、美分打印的为例来讲解 static factory method 的使用。注意美元的符号在前，而美分的符号在后。

1. 532 dollars => $532
2. 1030 cents => 1030 ¢

在面对这个问题时，抽象出了 UsdUnit 类，并提供两个 static factory method，以实现对 UsdUnit 实例生成的控制，并通过宏定义改善了 API 的可读性和方便性。

```
#include "base/Keywords.h"
#include "money/Amount.h"
#include <ostream>

DEFINE_ROLE(UsdUnit)
{
    static const UsdUnit& dollar();
    static const UsdUnit& cent();

    ABSTRACT(void format(std::ostream& oss, const Amount amount) const);
};

#define DOLLAR UsdUnit::dollar()
#define CENT   UsdUnit::cent()
```

```
#include "money/UsdUnit.h"

namespace
{
    struct dollar_class : UsdUnit
    {
        OVERRIDE(void format(std::ostream& oss, const Amount amount) const)
        {
            oss << "$" << amount;
        }
    };

    struct cent_class : UsdUnit
    {
        OVERRIDE(void format(std::ostream& oss, const Amount amount) const)
        {
            oss << amount << " ¢ ";
        }
    };
}

#define DEF_USD_UNIT(unit)              \
const UsdUnit& UsdUnit::unit()          \
{                                       \
    static unit##_class inst;           \
    return inst;                        \
}
```

```
DEF_USD_UNIT(dollar)
DEF_USD_UNIT(cent)
```

规则 6.1.12 通过私有构造函数强化不可实例化能力

此处提供了一个 String 的工具类，为了保证禁止生成实例的约束，提供了一个故意没有实现的默认构造函数，从而保证了编译时的安全性。

```
#include <vector>
#include <string>

struct StringUtil
{
    static std::vector<std::string> split
        (const std::string &text, char separator);

    static std::string wrap
        (const std::string &text, int wrapColumn = 80);

private:
    StringUtil();
};
```

但在实际操作中，常常不这么啰嗦。因为诸如类似的工具类，没有人会通过实例调用静态方法：

反例：

```
StringUtil().split("1,2,3", ',');
```

此时需要在编译安全性及其容忍冗余代码之间寻找平衡点。如果用户存在大概率地犯错的可能性，或做最保守的防御式编程，则应明确地拒绝；相反如果你的团队成员平均能力已经达到很高层次，日常 Code Review 的频率也比较高，则冗余的代码反而成为眼中刺。

是否需要将复制构造函数或拷贝赋值函数显式地声明为 private，道理是一样的。

规则 6.1.13 当一个类确定需要禁止被复制和赋值时，应明确地拒绝

为了防止其他人错误地进行值拷贝，常常需要禁止其复制和赋值行为，存在三种典型的技术：

```
#define DISALLOW_COPY(classname) \
private:                         \
    className(const className&)

#define DISALLOW_ASSIGN(classname) \
private:                           \
    className& operator=(const className&);

#define DISALLOW_COPY_AND_ASSIGN(className) \
    DISALLOW_COPY(classname)                \
    DISALLOW_ASSIGN(classname)
```

或 private 继承 Uncloneable 类:

```
class Uncloneable
{
protected:
    Uncloneable() {}
    ~Uncloneable(){}

private:
    Uncloneable(const Uncloneable&);
    Uncloneable& operator=(const Uncloneable&);
};
```

在 C++11 中，可以显式地 delete 掉相应的函数:

```
struct Test
{
    Test(const Test&) = delete;
    Test& operator=(const Test&) = delete;
};
```

但此规则只会在很少的情况下才被使用。因为在实际代码中，我们几乎都不应该进行值拷贝。除非该类的对象很容易被人错误地进行值拷贝，才需要显示地禁止其拷贝复制的行为。

规则 6.1.14 *避免创建不必要的对象*

对象创建和销毁是有代价的，尤其在关乎性能的对象创建和销毁时，需要特别地关注。

这时出现了很多技术来解决这方面的问题，对象池技术和 Flyweight 模式是最常见的技术，例如线程池，数据库连接池，网络连接池等等。

参考实例可参见 Joshua Bloch 所著的《Effective Java》, 规则同样适用于 C++。

6.2 继承与多态

规则 6.2.1 按接口编程

按接口编程是面向对象中最重要的原则之一，因为对业务抽象的接口往往体现了概念间的契约关系。但在 C++ 语言中，提供接口最让人繁琐的一件事情是：需要显式地提供一个 virtual 析构函数。

可以通过 DEFINE_ROLE 的宏定义来实现对接口的定义，从而可以消除子类对虚拟析构函数的重复实现。

```
namespace details
{
   template <typename T>
   struct Role
   {
      virtual ~Role() {}
   };
}

#define DEFINE_ROLE(type) struct type : ::details::Role<type>
```

```
DEFINE_ROLE(Runnable)
{
    ABSTRACT(void run());
}
```

规则 6.2.2 避免从并非设计为基类的类中继承

直到 C++11 标准才引入了类似于 Java 语言的 final 关键字，以便显式地阻止子类化。通常情况下，所有的 C++ 类都可以被继承。但存在如下几个场景，类应该设计为 final 类:

1. 并非为了安全地进行子类化而设计的类
2. 设计不可变类，保证多线程的安全性
3. final 类为编译器的优化提供更多的线索

如果一个类在设计之初，已经决定是 final 类，可以明确地告诉编译器。这里可以使用宏定义，即使你的编译器还不支持 C++11，通过 FINAL，表达对应的 virtual 函数不能再被覆写 (override) 或类不能被子类化，以便增强表现力。

```
#include <base/Config.h>

#if __SUPPORT_FINAL
# define FINAL final
#else
# define FINAL
#endif
```

因为历史原因，存在有些类并不是以子类化为目的而设计的，但是这些类往往拥有 public non-virtual 析构函数，当子类的析构函数需要释放特定资源时，将发生未定义的行为。

如果你试图继承 string 或 STL 的其他容器，将走上这条不归路。

规则 6.2.3 *为不具有多态性的基类提供* protected non-virtual *析构函数*

一个类可被子类化，但不具有多态性，明确地加上 protected non-virtual destructor 可改善设计的意图，提高表现力。

可惜的是，这条规则往往没有得到社区的普遍认识。当出现这样的场景，大家更习惯了让编译器自动生成一个默认的 public non-virtual destructor。所以，同样需要你在提高编译安全性与容忍冗余代码之间进行权衡和考虑。

标准库中的 std::input_iterator_tag, std::unary_function 等类就是不具有多态性质的基类。

规则 6.2.4 *考虑使用* virtual *函数声明* private，*而将* public *函数声明* non-virtual

此方法常被称为 NVI(non-virtual interface)，它是在 C++ 语言中实现 Template Method 模式的独特表现形式。Template Method 在父类中以一个 public 接口实现算法的骨架，并以 private virtual 挂接一个或多个回调钩子的成员函数供子类覆写。

规则 6.2.5 *区分* pure virtual, virtual, non-virtual *函数的语义*

当定义一个类时，清晰、准确的使用这三个特性，有助于其他人理解类设计的意图。

1. pure virtual: *为了让子类只继承其接口*
2. virtual: *让子类继承接口和一份默认的实现*

3. non-virtual: 让子类继承其接口和一份强制性的实现

规则 6.2.6 改写 virtual 函数时，必须显式地标示 override

在子类中改写虚函数时，提供显式的 override 将改善代码的可读性，更重要的是增强了编译时的安全性检查。

```
#include <cppunit/core/Test.h>

struct TestDecorator : Test
{
    explicit TestDecorator(Test& test);

private:
    OVERRIDE(int countTestCases() const);
    OVERRIDE(std::string getName() const);
    OVERRIDE(void run(TestResult& result);
    OVERRIDE(int getChildTestCount() const);

private:
    Test& test;
};
```

但可惜 override 关键字直到 C++11 才被列入标准，但可以提供 OVERRIDE 宏定义实现代码的可移植性，即使你的编译器不支持 C++11，也可以改善代码的表现力。

```
#include <infra/base/Config.h>

#if __SUPPORT_VIRTUAL_OVERRIDE
# define OVERRIDE(...) virtual __VA_ARGS__ override
#else
# define OVERRIDE(...) virtual __VA_ARGS__
#endif
```

规则 6.2.7 为多态基类提供 virtual destructor

何时需要给类增加 virtual destructor 常常让人抓狂，请记住两条规则：

1. As base class
2. Polymorphic

这两个条件必须同时满足，才可以为类增加 virtual destructor。一般地，如果一个类包含虚函数时，就为它增加 virtual destructor。

但需要注意的是，为子类化为目的，但没有运行时多态的语义时，Herb Sutter, Andrei Alexandrescu 建议显式地定义 protected non-vritual destructor。

规则 6.2.8 禁止在 constructor, destructor 中调用虚函数

在 constructor, destructor 调用期间，子类对象尚未构造，或已经销毁，所以在其父类的 constructor, destructor 中调用虚函数发生多态行为不是我们所期望的，所以禁止在 constructor, destructor 中调用虚函数。

规则 6.2.9 绝不重定义继承而来的 non-virtual 函数；也绝不重定义继承而来的缺省参数值

这是因为其重定义没有发生期望的多态行为，所以被禁止使用，以免产生反直觉的行为。

规则 6.2.10 避免遮掩继承而来的名称

这是 C++ 在嵌套作用域的名字隐藏机制在类作用域中的一个具体表现形式。当出现这些特殊的异常的情况时，往往能嗅探出不良命名和设计的坏味道。

规则 6.2.11 使用函数对象表示策略

策略模式是一种最常见的设计模式之一，C++ 语言中，常常使用函数对象、或定义多态策略的接口来实现策略的多态变化。

例如 std::sort 中实现的排序，使用 Comparator 方便用户定制比较规则。

```
template<typename Iterator, typename Comparator>
void sort(Iterator first, Iterator last, Comparator comp);
```

This page is intentionally left blank.

Premature optimization is the root of all evil.

– Donald Knuth

On the other hand, we cannot ignore efficiency.

– Jon Bentley

7 函数设计

7.1 函数

规则 7.1.1 避免过长，嵌套过深的函数实现

我讨厌长函数，犹如讨厌重复一样。长函数往往伴随着复杂的逻辑判断，过深嵌套逻辑。

但也必要硬性地规定团队函数不能超过固定的行，因为这个规则很容易被打破。如果大家都遵循良好的设计原则，养成良好的职责分离的设计的基本素养，实现出来的函数大多平均都在 5 行以内，或者更少。这绝对不是理想，已经存在众多的、典型的成功案例。

规则 7.1.2 只做一件事，并做好这件事

这是 SRP 在函数实现中的一个具体体现。一个函数只应该承担一个唯一的职责，如果函数名伴随 and，或将查询和命令混合往往是违背此原则的信号。

一般地，如果函数只是做了该函数名称下统一抽象层次上的几个小步骤，则函数还是只做了一件事情。函数就是将一个大一点的概念拆分成级别稍微低一点、并在同一个抽象层次的一系列步骤的过程。

规则 7.1.3 函数中的每一个语句都在一个相同的抽象层次上

如果函数中混杂不同抽象层次的代码，往往让人感到迷惑，理解代码的逻辑，需要读者陷入到细节之中而不能自拔。

Kent Beck 建议使用 Compose Method 分解长函数；Martin Flower 也建议使用 Extract Method 进行函数提取。Extract Method 最重要的就是识别出代码中的抽象层次，并梳理出主干，按照自顶向下的规则实现函数的。

函数提取常常要遵守如下 3 个基本原则：

1. 在同一个抽象层次

2. 给一个直观的，意图明确的好名字

3. 实现对称性

正如 Bob 大叔提到的那样: 程序就像是一系列的以 TO 起头的段落，每一段落都描述了当前抽象层次的逻辑，并引用下一抽象层次的，以 TO 开头的段落。

下例虽然以 Java 为例，但重点没有偏离，规则同样适用于 C/cpp。

```
public class List<E> {
   public void add(E element) {
      if (!readOnly) {
         int newSize = size + 1;
         if (newSize > elements.length) {
            Object[] newElements =
               new Object[elements.length + 10];
            for (int i = 0; i < size; i++)
               newElements[i] = elements[i];
            elements = newElements;
         }
         elements[size++] = element;
      }
   }
}
```

提取函数之后，使算法的骨骼显而易见。

```
public class List<E> {
   public void add(E element) {
      if (readOnly)
         return;

      if (atCapacity())
         grow();

      addElement(element);
   }
}
```

提取函数往往受到性能偏执狂的抨击，他们认为过多的函数提取将成为性能的主要瓶颈。2-8 原则将彻底击垮所有不靠谱的言论，与其纠结于函数调用的开销，不如花费更多的时间去优化和改善那些至关重要的算法和关键代码。

规则 7.1.4 检查参数的有效性

绝大部分的函数对于传递的参数都有限制，这时需要在函数执行之前完成参数的合法性校验。C/cpp 虽不能天然地支持“按契约设计”，但前置的参数检查将大大改善程序的健壮性。

规则 7.1.5 分离查询与指令

函数应该只拥有清晰明确的、唯一的职责。如果函数既修改对象状态，又返回对象的状态信息，则常常会导致混乱。产生副作用的指令操作，都是系统状态的一种变更，将产生时序上的耦合和顺序的依赖。尤其当查询和指令混合在一起的时候，查询函数的无副作用的优点将不复存在；如果一个函数名本身具有查询语义，但混合了指令操作，将产生更大的反直觉。

规则 7.1.6 使用 Null Object 替代空指针

校验空指针是一件及其繁琐的事情，优秀的程序员通过各种技巧绕开这些冗余的操作。例如参数通过引用传递，另外一种常见的手段就是 Null Object[1]。

规则 7.1.7 对于参数类型，返回值类型，优先使用接口类型

遵循按接口编程的良好习惯，是优秀设计师的基本素养。

规则 7.1.8 对于 bool 参数，优先使用两个元素的枚举类型

从直观上，

```
Thermometer::newInstance(CELSIUS);
```

要比

```
Thermometer::newInstance(true);
```

容易理解得多，但牺牲了一点点代码复用性。对于这个问题，可以通过内部的 private 函数来解决这个问题。也就是说，带 bool 参数的函数依然存在，只不过被 private，以便实现对枚举参数的函数的代码复用。

规则 7.1.9 永远不要导出具有相同参数数目的的重载方法

[1]参考 Martin Flower 的著作《重构，改善既有代码的设计》

重载是一种编译时的多态。只要函数具有相同名字，但参数数目，类型不同，即为重载。但滥用往往会误导使用 API 的程序员，尤其在具有相同参数数目的重载方法时，程序员需要清晰地知道所有类型的隐式转换规则，即其重载匹配规则，这无疑是一种没必要的负担。

```
struct ObjectOutputStream
{
    void write(bool);
    void write(char);
    void write(short);
    void write(int);
    void write(long);
    void write(float);
    void write(double);
};
```

面对疑难的时候，抛弃重载往往能得到更不容易犯错的解决方案。

```
struct ObjectOutputStream
{
    void writeBool(bool);
    void writeChar(char);
    void writeShort(short);
    void writeInt(int);
    void writeLong(long);
    void writeFloat(float);
    void writeDouble(double);
};
```

规则 7.1.10 使用 explicit 显式地禁止类型的隐式转换

隐式类型转换往往无声无息地被执行，通过 explicit 便能通过编译器清晰地捕获的所有隐式转换的事件，以便供程序员进一步决策。

规则 7.1.11 避免实现火车失事的代码

```
String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

本例虽然使用 Java 语言编写，但这不影响对火车失事问题的理解。如果其调用链如果某一环节出现了问题，则整个调用将出现失败。这类串联的调用违反了 Demeter 法则，ctxt 对象包含了多个选项，每个选项中存在一个临时目录，每个目录都有一个绝对路径，所有的知识都毫无保留地暴露给了用户。

仔细分析一下代码，再看看其中一个模块是如何使用这个 outputDir 的。

```
String outFile = outputDir + File.SEPERATOR + className + ".class";
BufferedOutputStream bos = new BufferedOutputStream(new
  FileOutputStream(outFile));
```

所有的一切都是为了创建指定名称的临时文件，理想的实现应该将所有本应该隐藏的知识归并到 ctxt 中实现。

```
BufferedOutputStream bos = ctxt.createScratchFileStream(className);
```

7.2 重载操作符

规则 7.2.1 重载运算符必须保持原有操作符的语义

如果重载运算符改变了程序员对固有知识的理解，将加大放错的几率。

规则 7.2.2 谨慎地使用转换运算符

转换运算符因为其隐式转换而变得非常神秘，需谨慎使用。并并非在任何场景都不使用转换操作符，在合适的情况下使用装换运算符，能够得到更为简洁的代码。

规则 7.2.3 优先使用前置 ++operator

当重载了 ++operator 的类对象，例如迭代器，更提倡使用 ++operator，这是提升效率的一种举措。

规则 7.2.4 使用 operator*, operator-> 实现类的包装或修饰

operator*, operator-> 操作符是提供包装和修饰功能的特殊工具，是一种典型的间接技术[1]。它的最大优势是为用户提供良好的、人性化间接操作接口。

std::unique_ptr, std::shared_ptr 等智能指针，最关键的也是重载了这两个操作符，从而实现了操作智能指针犹如操作原始指针一般。

Placment 本身也是一个包装器，只不过它修饰的是一块原始的内存。通过 operator*, operator-> 可方便地操作寄存在原始内存上的 T 对象。

[1]软件工程有一条黄金定律：任何问题都可以通过增加一个间接层来解决。

AutoMsg 也是一个较为经典的例子，AutoMsg 将消息对象的构造搬迁到了堆上进行初始化，从而避免了由于消息过大而导致的栈溢出的风险。当函数调用出栈时，内存自动释放。

```
template <typename Msg>
struct AutoMsg
{
    AutoMsg() : msg(new Msg)
    {}

    ~AutoMsg()
    {
        delete msg;
    }

    Msg& operator*()
    {
        return *msg;
    }

    Msg* operator->()
    {
        return msg;
    }

private:
    Msg* msg;
};
```

```
#include <money/domain/Account.h>
#include <money/msg/TransMoneyMsg.h>
#include <base/Asserter.h>

struct RemoteAccount
{
    Status transMoneyTo(const Account& to)
    {
        AutoMsg<TransMoneyMsg> msg;

        ASSERT_SUCC_CALL(buildMsg(*msg));
        ASSERT_SUCC_CALL(sendMsg(to, *msg));

        return SUCCESS;
    }

private:
    Status buildMsg(TransMoneyMsg&)
    {
        // no implement
    }

    Status sendMsg(const Account& to, const TransMoneyMsg& msg)
    {
        // no implement
    }
};
```

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

- C.A.R. Hoare

8 整洁测试

8.1 TDD

规则 8.1.1 TDD 遵循三定律

1. 在编写不能通过的测试前，不可编写生产代码
2. 只可编写刚好无法通过的测试，不能编译也算不通过
3. 只可编写刚好足以通过当前失败测试的生产代码

关于测试驱动请参考 Kent Beck 的著作《Test-Driven Development, by example》。

规则 8.1.2 测试名称遵循 Given-When-Then

这是流行的 BDD，行为驱动测试命名规范，当阅读测试用例时，或当测试失败时，这样的命名风格非常有利于行为的描述。

摒弃原始的 TestXXX 的风格吧，因为它不能清晰表达 TDD 的意图。

规则 8.1.3 每一个测试一个概念

这是 SRP 在测试用例中的体现，当测试用例失败的时候，能够清晰地知道测试失败的原因。

一个概念往往只会有一个断言来描述，出现数目众多的断言往往违背了此规则。

规则 8.1.4 将具有相同测试环境的用例放在同一个测试套件中

将构造测试环境及清理测试环境的代码分别放在 Setup 及 Teardown 中，可以消除重复的代码，实现代码的最大可复用性。

规则 8.1.5 测试应该像文档一样清晰

测试用例是理解系统行为的最佳途径，也是最实时、最权威的文档。

规则 8.1.6 测试更应该是像一个例子

测试不仅仅为了测试而测试，更重要的是对系统行为的描述。

规则 8.1.7 测试用例中不应该存在复杂的循环、条件控制语句

测试用例堆可读性的要求非常高，如果出现大量的循环、条件控制语句，将大大地损害了用例的可读性。一般地，测试用例应该是有若干条陈述句所组成，越简单越好。

规则 8.1.8 设计面向特定领域的测试语言是值得的

根据领域问题，设计特定领域描述语言 DSL，将改善用例的可读性和维护性。