C++ Programming Practices

Guidelines, Rules, and Patterns

刘光聪

This page is intentionally left blank.

目录

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

1 Design Priciples

1.1 SOLID

SOLID 完整地描述是由 Robert C. Martin 在其脍炙人口的著作《敏捷软件开发，原则、模式与实践》中阐述的，成为了面向对象设计的最基本的原则。

1. SRP, The Single Responsibility Principle
2. OCP, OCP The Open Closed Principle
3. LSP, The Liskov Substitution Principle
4. ISP, The Interface Segregation Principle
5. DIP, The Dependency Inversion Principle

1.2 Simple Design Principles

这是 XP 倡导的最基本的四个简单设计原则，其重要性依次进行排列。

1. runs all the tests
2. says everything OnceAndOnlyOnce
3. expresses every idea that we need to express, Self-documenting code
4. has no superfluous parts

1.3 Orthotropic Design Principles

这是袁英杰在其 OO 训练营中阐述的理论和指导原则，并对作者产生了巨大的影响，正交设计四原则成为了作者在设计中最具有指导意义的原则之一。

1. 消除重复
2. 分离关注点
3. 最小依赖
4. 向稳定的方向依赖

1.4 Others

1. DRY, Don't repeat youself
2. Law of Demeter
3. YAGNI, You ain't gonna need it
4. KISS, Keep it simple, stupid
5. CQS, Command-Query separation
6. Hollywood Principle, Don't call us, we'll call you

Do the simplest thing that could possibly work.

- Kent Beck

2 Physical Design

2.1 Design Principles

原则 2.1.1 自满足

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 RrmSrb1AdmitAction.h 头文件。

```
#ifndef INCL_DCM_RrmSrb1AdmitAction_H
#define INCL_DCM_RrmSrb1AdmitAction_H

struct RrmSrb1AdmitAction : SyncAction
{
    OVERRIDE Status exec(const TransactionInfo&);
};

#endif
```

创建对应的实现文件 RrmSrb1AdmitAction.cpp

```
#include "rrm/actions/RrmSrb1AdmitAction.h"
```

将自身头文件的进行包含，并置在实现文件的第一行，是验证头文件自满足的最佳方式。

如上例所示，RrmSrb1AdmitAction 类的定义存在对父类 SyncAction 的编译时依赖，为了让其自满足，必须包含父类的头文件。

```
#ifndef INCL_DCM_RrmSrb1AdmitAction_H
#define INCL_DCM_RrmSrb1AdmitAction_H

#include "trans-dsl/action/SyncAction.h"

struct RrmSrb1AdmitAction : SyncAction
{
    OVERRIDE Status exec(const TransactionInfo&);
};

#endif
```

原则 2.1.2 单一职责

这是 SRP，Single Reponsibility Priciple 在设计头文件时的一个具体运用。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，重编译成为一件高概率的事件。

如示例代码，将 SRrmSrb1AdmitAction, RrmErabSetupAdmitAction 同时定义在一个头文件中，将违背该原则。

```
#ifndef INCL_DCM_RrmAdmitAction_H
#define INCL_DCM_RrmAdmitAction_H

#include "trans-dsl/action/SyncAction.h"

struct RrmSrb1AdmitAction : SyncAction
{
    OVERRIDE Status exec(const TransactionInfo&);
};

struct RrmErabSetupAdmitAction : SyncAction
{
    OVERRIDE Status exec(const TransactionInfo&);
};

#endif
```

原则 2.1.3 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用的降低编译时依赖的技术。

```
#ifndef INCL_EventHandler_H
#define INCL_EventHandler_H

#include "base/Keywords.h"
#include "base/Status.h"
#include "fw/event/Event.h"

INTERFACE(EventHandler)
{
    virtual Status handleEvent(const Event&) = 0;
};

#endif
```

如示例代码，定义了一个 EventHandler 的接口，对 Event 仅仅是一个声明依赖，并没有必要实现对 Event.h 的包含，前置声明是解开这类编译依赖的钥匙。

但对于本例，对 typedef WORD32 Status; 的依赖，以及对宏 INTERFACE 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。

```
#ifndef INCL_EventHandler_H
#define INCL_EventHandler_H

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

struct Event;

INTERFACE(EventHandler)
{
    virtual Status handleEvent(const Event&) = 0;
};

#endif
```

在选择 #include 和前置声明的时候，很多程序员感到迷茫，其实规则很简单。如果一个头文件依赖的名称仅作为指针、引用、返回值，参数无需包含头文件，前置声明即可；

相反，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。主要包括如下几种场景。

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

原则 2.1.4 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。

最可怕的就是将所有的实体都 public，这无疑是一种自杀式做法。我们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。

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。

```
#ifndef SIMPLEASYNCACTION_H_
#define SIMPLEASYNCACTION_H_

#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);

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

private:
   virtual void doKill(const TransactionInfo&, const Status) {}

private:
   EventHandlerRegistry registry;
};

#endif
```

2.2 Inline

规则 2.2.1 *头文件中不应出现任何的* inline *函数实现。*

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有点累赘和麻烦。

1. virtual *析构函数*

2. 空的 virtual 函数实现

规则 2.2.2 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，往往只有一个。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，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，当滥用单态，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。

```
#include "base/Status.h"
#include "base/BaseTypes.h"
#include "base/Keywords.h"
#include "cell/LteCell.h"
#include <vector>

struct LteCellRepository
{
    static LteCellRepository& getInstance();

    Status add(const WORD16 cellId);
    Status release(const WORD16 cellId);
    Status modify(const WORD16 cellId);

private:
    typedef std::vector<LteCell> Cells;
    Cells cells;
};
```

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

此时，可以通过一系列措施降低编译时依赖的问题。一个直觉上解决方案就是将依赖置入到源文件中，解除 LteCellRepository 的编译时依赖。

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

struct CellVisitor;

INTERFACE(LteCellRepository)
{
    static LteCellRepository& getInstance();

    virtual Status add(const WORD16 cellId) = 0;
    virtual Status release(const WORD16 cellId) = 0;
    virtual Status modify(const WORD16 cellId) = 0;
};
```

```
#include "cell/LteCellRepository.h"
#include "cell/CellVisitor.h"
#include "cell/LteCell.h"
#include <vector>

namespace
{
    struct LteCellRepositoryImpl : LteCellRepository
    {
        OVERRIDE Status add(const WORD16 cellId)
        {
            // inline implements
        }

        OVERRIDE Status release(const WORD16 cellId)
        {
            // inline implements
        }

        OVERRIDE Status modify(const WORD16 cellId)
        {
            // inline implements
        }

    private:
        typedef std::vector<LteCell> Cells;
        Cells cells;
    };
}

LteCellRepository& LteCellRepository::getInstance()
{
    static LteCellRepositoryImpl inst;
    return inst;
}
```

此处，LteCellRepositoryImpl 类实现时，所有实现都进行 inline 实现，是否真正地进行 inline 就让编译器自行决定吧。懒惰的我一直信仰着 Once & Only Once 的真理。

注：此处为了简化实现和举例，使用了 STL 中的 std::vector。

规则 2.2.3 *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。

此刻，请记住匿名命名空间也是一种重要的信息隐藏技术。

2.3 struct VS. class

规则 2.3.1 *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。这体现在定义时和继承时。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。

我讨厌冗余和重复，在定义接口和继承时，冗余 public 修饰符总让人不舒服。

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

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

更重要的是，我确信“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，class 的特性正好与我的期望背道而驰[1]，至此我深深地被 struct 所痴迷，不能自拔。

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，所有的前导声明都需要修改。

规则 2.3.2 *定义 C 风格的结构体时，摒弃* struct tag *命名风格。*

struct tag 严重地阻碍了结构体的前置声明。

[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。

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

为了兼容 C[1]，并为结构体前置声明提供便利，如下解法也许是最合适的了。

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

规则 2.3.3 *考虑使用 PIMPL 降低编译时依赖*

```
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。

[1]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 struct T_Cell *pcell，而 C++ 没有这方面的要求。

2.4 template

规则 2.4.1 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。

```
#ifndef INCL_DCM_OSS_SENDER_H
#define INCL_DCM_OSS_SENDER_H

#include "base/Status.h"
#include "Pub_TypeDef.h"
#include "Pub_Oss.h"
#include "OSS_Comm.h"
#include "Pub_CommDef.h"
#include "base/Assertions.h"

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

    template <typename MSG>
    Status send(WORD16 eventId, const MSG& msg)
    {
        DCM_ASSERT_TRUE(OSS_SUCCESS == OSS_SendAsynMsg(eventId, &msg, \
          sizeof(msg), commType, (PID*)&pid);
        return DCM_SUCCESS;
    }

private:
    const PID& pid;
    BYTE commType;
};

#endif
```

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。

```
#ifndef INCL_DCM_OSS_SENDER_H
#define INCL_DCM_OSS_SENDER_H

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

struct PID;

struct OssSender
{
    OssSender(const PID& pid, BYTE commType)  : pid(pid), commType(commType)
```

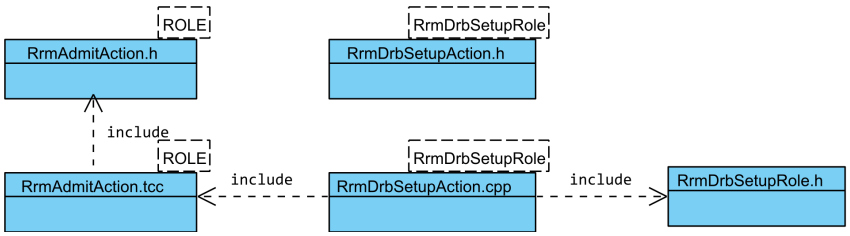

图 2-1 Explicit Template Instantiated

```
    {
    }

    template <typename MSG>
    Status send(WORD16 eventId, const MSG& msg)
    {
        return send(eventId, (const void*)&msg, sizeof(MSG));
    }

private:
    Status send(WORD16 eventId, const void* msg, size_t size);

private:
    const PID& pid;
    BYTE commType;
};

#endif
```

识别哪些与泛型相关，哪些与泛型无关解开此类的编译时依赖的钥匙。

规则 2.4.2 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。可惜 export 模型受编译技术实现的调整，最终被 C++ 11 标准抛弃。

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效果的。

如图2-1（第12页）所示，DrbSetupAdmitAction.h 仅仅对 RrmAdmitAction.h 产生依赖; 特殊地，DrbSetupAdmitAction.cpp 没有产生对 DrbSetupAdmitAction.h 的依赖，相反对 RrmAdmitAction.tcc 产生了依赖。

当存在很多诸如 DrbSetupAdmitAction.h 的子类时，编译时依赖将被彻底解除。

```
// RrmAdmitAction.h
#include "fw/action/SyncAction.h"

template <typename ADMIT_ROLE>
struct RrmAdmitAction : SyncAction
```

```
{
    OVERRIDE Status exec(const TransactionInfo&);
};
```

```
// RrmAdmitAction.tcc
#include "rrm/actions/RrmAdmitAction.h"
#include "dcm_inst/UeInstanceHelper.h"
#include "cell/UeCellObject.h"
#include "base/Assertions.h"

template <typename ADMIT_ROLE>
Status RrmAdmitAction<ADMIT_ROLE>::exec(const TransactionInfo& trans)
{
    ADMIT_ROLE* role = getCurrentCell(trans);
    DCM_ASSERT_VALID_PTR(role);

    return role->admit();
}
```

```
// DrbSetupAdmitAction.h
#include "rrm/actions/RrmAdmitAction.h"

struct RrmDrbSetupRole;

struct DrbSetupAdmitAction : RrmAdmitAction<RrmDrbSetupRole> {};
```

```
// DrbSetupAdmitAction.cpp
#include "rrm/role/RrmDrbSetupRole.h"
#include "rrm/actions/RrmAdmitAction.tcc"

template struct RrmAdmitAction<RrmDrbSetupRole>;
```

规则 2.4.3 子类化优于 typedef

```
#include "rrm/actions/RrmAdmitAction.h"

struct RrmDrbSetupRole;

struct DrbSetupAdmitAction : RrmAdmitAction<RrmDrbSetupRole> {};
```

这样做唯一的理由是，当仅仅出现对 DrbSetupAdmitAction 的依赖时，前置声明成为可能；此外，你可以重写 RrmAdmitAction 中的虚函数，实现运行时多态。

但如果实现的是 typedef，除了包含头文件之外，别无他法，无疑增加了不必要的编译时依赖。

```
#include "rrm/actions/RrmAdmitAction.h"

struct RrmDrbSetupRole;

typedef RrmAdmitAction<RrmDrbSetupRole> DrbSetupAdmitAction;
```

Write programs for people first,
computers second.

- Steve McConnell

3 Naming

命名是一件具有浓厚艺术气息的技艺。良好的命名将改善代码的表现力，多查询字典、多与你的结对伙伴沟通是一种改善命名的好习惯。

3.1 Baby Names

规则 3.1.1 遵守统一的命名规范

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

表 3.1 命名规范

规则 3.1.2 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；返回值为 bool，加上 is, has, can, should, need 将会增强语意。

以 bool 变量为例，如下例代码所示。

```
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

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

规则 3.1.3 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3.2（第16页）所示。

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 3.2 Colorfull Words

规则 3.1.4 程序中每个实体都应该有一个 Intention-Revealing 名称。

```
public vector<vector<int> > getThem() {
    vector<vector<int> > list1;
    for (auto x : theList)
      if (x[0] == 4)
        list1.add(x);
    return list1;
}
```

1. vector<vector<int> > 的语法令人抓狂
2. getThem 让人看不清楚你的意图
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:
    vector<int> states;
};

typedef vector<Cell> Cells;
```

重构之后的效果，抽象和可读性得到了改善。

```
public Cells getFlaggedCells() {
    Cells flaggedCells;
    for (auto cell : gameBoard)
      if (cell.isFlagged())
        flaggedCells.add(cell);
    return flaggedCells;
}
```

注：本例使用了 C++ 11 的 for-each 语法，简化迭代器的操作。

规则 3.1.5 使用注释不如花费更多时间给实体取一个好名字。

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

```
int elapsedTimeInDays;
```

虽然我更加习惯用代码阐述意图，但有时我也会增加必要的注释使代码更加容易被别人理解和维护。

尤其在进行位运算时，提供比特位的内存解释代码，将大大改善代码的可读性；其次，代码中存在特殊的陷阱，实现手法或特殊情况，注释此时变得非常宝贵。

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

规则 3.1.6 Noise Words are Redundant，消除噪声后将得到一个更加精准的名字。

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

表 3.3 Noise Words are Redundant

规则 3.1.7 使用 Domain 的名称，将直白地表明你的设计，我们鼓励这样的命名。

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

3.2 Hungarian Notation

规则 3.2.1 *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，编译器也记得并强制使用类型；而且，人们趋于使用更小的类，更短的函数，让每一个变量定义都在视野范围之内；另外，程序员得到了 IDE 强有力的支撑，匈牙利命名反而变成了一种噪声和肉刺，是时候摒弃它了。

规则 3.2.2 *摒弃成员前缀，或后缀。*

与其写形如 m_name 的成员变量，还不如花费更多的时间将类分解；当类足够小，职责单一，使用现代 IDE，前缀就像一根刺，让你的眼睛不舒服。

规则 3.2.3 *摒弃接口的前缀*

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

struct TransactionInfo;

INTERFACE(IAction)
{
    virtual Status exec(const TransactionInfo&) = 0;
}
```

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

Make it work, make it right,
make it fast.

- Kent Beck

4 General Programming

4.1 Code Style

规则 4.1.1 团队应该统一代码风格

团队代码拥有一种最权威的风格，犹如一个人写的一样。否则代码因缺乏统一整齐的花括号，或者存在长度不一致的 TAB，使得代码变得层次不齐。

存在多种经典的代码风格，它们都拥有各自的特点和优势，团队应该统一地选择其中一种即可。

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问题。

规则 4.1.2 拒绝实现巨文件

巨大的头文件最大的一个问题就是产生了难以忍受的编译时依赖，这种往往是由 C 遗留的一些陋习所致，拒绝它的诱惑吧。

巨大的实现文件虽然没有编译时依赖的问题，但它给人带来巨大的恐惧感；此外在一个 100000 行的源文件中进行重构简直就是一件痴人说梦的事情；在多人同时修改此文件的概率大大加大，使用配置管理工具时冲突不断。

多好行是最适合的呢，没有标准，犹如一个函数应该有多少行一样的问题；只有一个高内聚、低耦合的原则可参考；但事实证明，往往满足了这个原则的时候，文件就变得短小精干了。

4.2 Clean Comment

规则 4.2.1 *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，在调整读者的智商；例如下面的第一个注释，谁不知道它是一个构造函数呢？

是的，它们的存在的确能产生格式化很漂亮的 document，但这样的 document 又有多少人真正需要呢？

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

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

还有一种就是误导性的注释，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。

还有一种注释我们经常遇到：日志型注释。修正一个 bug 时，都需要在函数头的注释表中增加一条记录，放弃这种好习惯吧，将这份精细的工作交给源代码控制系统吧。

还有一种注释，当我遇到时，就会产生一种莫名的冲动将其消灭。在花括号后面的//end if, //end while, // end try，它们的出现就是一个提取函数的鲜明信号。

代码签署问题，至于我从来不会将自己的名字留在代码中，也许只有代码巨匠才敢做这件事情。事实上，留了名字的代码往往只会给后来人唯一的一个提示就是：这个代码只有我一个人能维护，别抢我的饭碗。

已经被注释掉的代码，应该立即删除。因为从源代码控制系统中追回这份被删除的代码是很容易的。

规则 4.2.2 *在需要注释的时候，一定要加注释*

法律信息注释，这是因为公司版权的保护不得不让程序员增加这些注释。这类

问题幸好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[1]；

对代码无法明确的意图需要增加注释；对于那些需要程序员花费更多思考才能理解的代码也需要增加注释，例如对协议包的注释，正则表达式的注释，位移操作的注释等。

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

当在代码中存在反直觉，存在可能陷阱的代码也需要增加注释。这类情况，首先应该考虑能否消除这样的陷阱，但有时候是很难做到，注释可能是最后的一根救命稻草了。

4.3 General Rules

规则 4.3.1 绝不使用全局变量

全局变量犹如 goto 一样臭名昭著，但程序员常常被全局变量所诱惑，因为它的确容易被实现。但是它带来了破坏性远远大于其益处。作为一个团队，应该拥有统一的编码规程，禁止使用全局变量应该成为团队中大家必须遵守一条法文。

太苛刻了，但从实践情况上了，这样的要求并没有想象中那么苛刻，总会找到一种比全局变量更好的解决方案。

在最坏的情况下，当需要给系统唯一存在的概念进行描述，也不应该定义全局变量，而应该使用单态进行包装。

规则 4.3.2 只有在简单逻辑的情况下鼓励使用?: 条件表达式。

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

这样的语句鼓励重构为:

[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

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

虽然道理简单，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。犹如常常看到代码中会有如下的语句一样，让人哭笑不得。

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

但是，当表达式变得非常复杂时，此时，if-else 可能是更好的选择了。

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

规则 4.3.3 *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。*

```
bool contains(const std::string* str, const string* substr)
{
    if (str == NULL || substr == NULL)
        return false;

    if (substr->empty())
        return true;

    ...
}
```

可以想象，如果不进行提前返回，上述处理的表达效果将大打折扣。注，此处故意没有使用 const 引用，以便举例。

虽然在代码阅读过程中 continue 常常打断了人的思维，但在这种情况下，continue 相反能给人一种“跳过此项”的意图，代码嵌套逻辑得到了改善。

是的，单一出口原则在现代 C++ 中程序中已经变得过时了，毕竟 C++ 提供了更为精细的处理方式来完成遗留的清理工作：析构函数[1]。

规则 4.3.4 *擅用解释型变量，或提供意图明确的、内联的查询函数。*

解释型变量的重构手法因为内联的查询函数的威力所大大折扣，往往没有后者运用广泛。

[1]Java 也提供了类似的机制，即 try-finally 机制，从 JDK1.7 加入 Closeable 接口出现之后，try-finally 机制变得更加实用了。

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

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

注意，split 函数将字符串按照指定的正则表达式进行拆分。

规则 4.3.5 如果需要精确答案，请避免使用 float 和 double

由于二进制浮点运算只是一种快速的近似的数值运算计算，所以在精度上可能与想象的有一定误差。常见的做法是通过扩大倍数，将浮点数运算转换为整数运算。

规则 4.3.6 复用类库，复用既有代码。

复用类库，复用既有代码，而不是重新造一个轮子，这给程序员提出了一个很大的挑战。就像学习 ruby 一样，入门简单，但真正的精通，最后比的还是对库的熟悉运用程度。

This page is intentionally left blank.

Controlling complexity is the essence of computer programming

- Brian Kernighan

5 Immutability

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。

5.1 const

规则 5.1.1 使用 const 替代 #define 常量的宏定义

使用 const 替代宏常量的理由应该是众所周知，但往往受 C 语言根深蒂固的习惯常常被人遗忘。

规则 5.1.2 当成员函数具有查询语义时使用 const 进行声明

此处具有查询语义的函数，泛指所有对对象状态产生副作用的函数，但未必都以 get, is 开头。

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

struct RbId
{
    explicit RbId(BYTE id);

    bool isSrbId();
    bool isDrbId();
    BYTE getValue();

    bool operator==(const RbId&);
    bool operator!=(const RbId&);

private:
    BYTE id;
};
```

我们提倡诸如 RbId 类对业务逻辑进行封装，即使它的内存本质上仅仅是一个整数而已。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。

此例唯一不足的是对于那些具有查询语义的函数并没有声明为 const。const 的存在不仅仅是为了提高编译时的安全性检查，更重要的是提供了良好设计的信号。

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

struct RbId
{
    explicit RbId(BYTE id);

    bool isSrbId() const;
    bool isDrbId() const;
    BYTE getValue() const;

    bool operator==(const RbId&) const;
    bool operator!=(const RbId&) const;

private:
    BYTE id;
};
```

规则 5.1.3 *尽量以 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实例。

规则 5.1.4 *当传递内置类型，迭代器及函数对象时，比较适合 pass-by-value。但使用 const 修饰，往往能够改善 API 的可读性。*

已知 Status, ActionId 分别是 WORD32, WORD16 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。

```
#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 5.1.5 *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。*

[1]关于对称性，请参考 Kent Beck 的著作《实现模式》

```
#include "base/Status.h"
#include "base/BaseTypes.h"
#include <stddef.h>

struct Gid
{
    bool isValid() const;

    Status sendToMe(WORD16 eventId, const void* msg, size_t size) const;

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

    static const Gid getSelfGid();
    static const Gid valueOf(WORD16 value);
};
```

静态工厂方法 valueOf, getSelfGid 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。

规则 5.1.6 不能返回局部对象的引用或指针。

再返回值类型和引用类型常常会困扰着我们。其实规则非常简单，返回值对象当且仅当需要创建新的对象时；此时企图返回引用或指针将返回局部对象的引用或指针，从而造成可怕的运行时异常。

更有甚者，此类错误编译器往往给出警告，而非错误。此刻需要程序员要非常熟练地控制这些技巧。

5.2 Immutable Class

规则 5.2.1 当设计 Value Object 时，常常实现为 Immutable Class

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，提倡重用对象实例

设计不可变类，需遵循下面几天规则:

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. 常常不能被复制，但绝不允许赋值

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。

以 RbId 的实现来讲解不可变类设计的几个要点。

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

struct RbId
{
    explicit RbId(BYTE id);

    bool isSrbId() const;
    bool isDrbId() const;
    BYTE getValue() const;

    bool operator==(const RbId&) const;
    bool operator!=(const RbId&) const;

private:
    const BYTE id;

private:
    DISALLOW_COPY_AND_ASSIGN(RbId);
};
```

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

- Kent Beck

6 Class Design

6.1 Construction, Destruction

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，它与 SRP(Single Reponsibility Principle) 相违背。程序员几乎都会碰到这样的巨类，搜索其成员变量的修改点是一件痛苦的事情；成员变量在 20-30 个成员函数中穿梭，变得像全局变量一样难理解。不仅如此，修改这个巨类，冲突常常发生，修改这样的高度耦合的类需要高超的技能。

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，在此不深入探讨此方面的内容[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，尤其面对复杂的 C++ 语言时。完全且深刻理解其初始化规则是 C++ 程序员基本素质之一。

C++ 使用构造函数完成初始化，而构造函数使用初始化列表完成初始化的。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；如果对应的父类或成员变量没有默认构造函数，则需显式地调用调用特定的带参构造函数，否则编译失败。如果没有显式地调用本类的成员变量，规则也类似。

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。

规则 6.1.3 手工初始化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其初始化。

[1]详细内容可参考 Michael C. Feathers 所著的《Working Effectively with Legacy Code》，中文版名为《修改代码的艺术》

TransData 必须实现其对基本数据类型的成员变量的初始化，否则它们就是未定义的，其行为未定义。

```
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 currentIndex :1;
   unsigned char :4;
};

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

规则 6.1.4 *在初始化列表中实现对非内置数据类型的初始化*

非内置数据类型指类类型，如果它们没有在初始化列表中初始化，则编译器默认调用其默认构造函数 (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重新赋值的语义。如果将非内置数据类型放在构造函数体内，则编译器首先调用默认构造函数，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，也不是我们期望的。

规则 6.1.5 *初始化列表顺序与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，往往出现先调用初始化顺序依赖的问题。

规则 6.1.6 *确保构造函数对每一个成员进行初始化*

除非你确认编译器能够正确地调用对应成员的默认构造函数，否则需要显式地调用其带参构造函数完成其初始化。

规则 6.1.7 *当一个类需要禁止被复制和赋值时，应明确地拒绝*

为了防止不应该的值拷贝，常常需要禁止其复制和赋值行为，常常使用两种技术；

```
#define DISALLOW_COPY_AND_ASSIGN(className) \
private:                                    \
    className(const className&);            \
    className& operator=(const className&);
```

或 private 继承 Uncloneable 类。

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

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

规则 6.1.8 使用 RAII 进行资源的安全管理

RAII(Resource acquisition is initialization) 是一种有用的技术，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.
}

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

规则 6.1.9 绝不 public 内部的成员变量

public 数据结构违背了信息隐藏机制，这是一种典型的设计坏味道。

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

如图6-1（第32页）所示，面临众多重载的构造函数时，你知道需要调用哪一个吗？但通过提供具有意图明确的 static factory method，并将对应的构造函数 private，将极大地改善 API 的可读性，是良好设计的榜样。

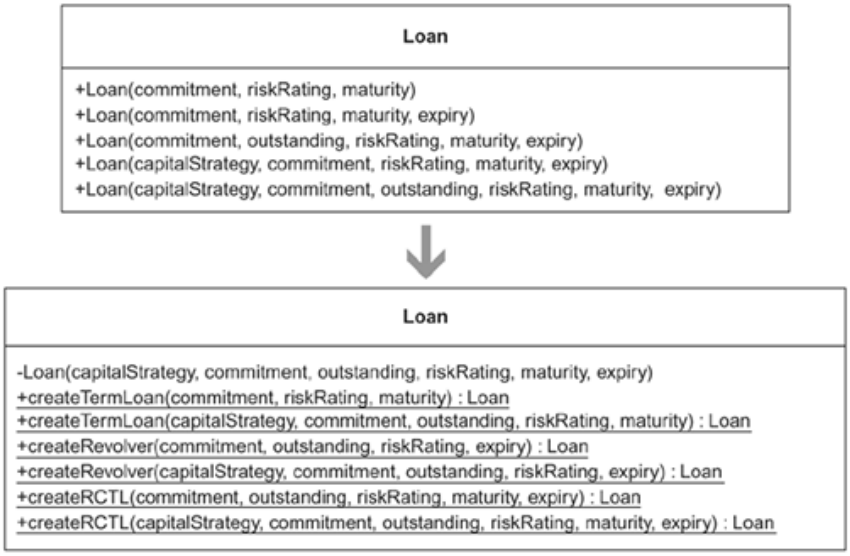

图 6-1 static factory method

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，有利于复用对象，提升性能

3. 返回类型可以是任何子类型的对象

再以一个具体的实例来讲解 static factory method 的使用。按照美元的原始单位进行打印。注意：dollar 的符号在前，而 cent 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 DollarUnit 类，并提供两个 static factory method，以实现对 DollarUnit 实例生成的控制，并通过宏改善了 API 的可读性和方便性。

```
#ifndef INCL_DollarUnit_H_
#define INCL_DollarUnit_H_

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

//////////////////////////////////////////////////////////////////////////
ABSTRACT(DollarUnit)
{
    static const DollarUnit& dollar();
    static const DollarUnit& cent();

    virtual void format(std::ostream& oss, const Amount& amount) const = 0;

protected:
    explicit DollarUnit(const Amount& convertionFactor);

private:
    const Amount convertionFactor;
};

//////////////////////////////////////////////////////////////////////////
#define DOLLAR DollarUnit::dollar()
#define CENT   DollarUnit::cent()

#endif
```

```
#include "money/DollarUnit.h"
#include "base/Keywords.h"
#include <string>

namespace
{
    struct Dollar : DollarUnit
    {
        Dollar(const Amount& convertionFactor)
         : DollarUnit(convertionFactor) {}

    private:
        OVERRIDE void format(std::ostream& oss, const Amount& amount) const
        {
            oss << "$" << amount;
        }
    };
```

```
    struct Cent : DollarUnit
    {
        Cent(const Amount& convertionFactor)
         : DollarUnit(convertionFactor) {}

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

const Amount CENT_CONV_FACTOR   = 1;
const Amount DOLLAR_CONV_FACTOR = 100 * CENT_CONV_FACTOR;

const DollarUnit& DollarUnit::dollar()
{
    static Dollar dollar(DOLLAR_CONV_FACTOR);
    return dollar;
}

const DollarUnit& DollarUnit::cent()
{
    static Cent cent(CENT_CONV_FACTOR);
    return cent;
}

DollarUnit::DollarUnit(const Amount& convertionFactor)
 : convertionFactor(convertionFactor)
{
}
```

规则 6.1.12 通过私有构造函数强化不可实例化能力

以早期 boost 实现单态的技术为例讲解这个问题。

```
template<typename T>
struct Singleton
{
    static T& getInstance()
    {
        static T instance;
        return instance;
    }

private:
    Singleton& operator=(const Singleton&);
    Singleton(const Singleton&);

protected:
    Singleton() {}
};

#define SINGLETON(object) struct object : Singleton<object>
```

规则 6.1.13 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 Inheritance, Polymorphism

规则 6.2.1 优先提供抽象的接口定义，并按接口编程

按接口编程是面向对象中最重要的原则之一。对业务抽象的接口往往表现为明确的契约关系，在 C++ 语言中，虽然没有接口这样的概念，但已经存在成熟的设计方案。

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

#define INTERFACE(type) struct type : ::details::Interface<type>
```

使用上述提供的 INTERFACE 宏，在定义一个接口时，可以省去对虚拟析构函数的重复实现。

```
INTERFACE(Runnable)
{
    virtual void run() = 0;
}
```

规则 6.2.2 避免从并非设计为基类的类中继承

C++ 并没有提供像 Java 语言一样的 final 关键字来组织类被继承的机制。通常情况下，所有的 C++ 类都可以被继承。

但是有些类最初并不是以基类的角度进行设计的，这种类的往往却拥有 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。

是否拥有何种技术提供类似 Java 语言一样的 final 关键字的机制呢，以便向读者透露出此类不可被继承的设计意图？其实 C++ 11 标准已经将 final 列为关键字，支持了这项语义。在此之前，也有类似的模拟技术实现 final 的功能，但实现复杂了一点，在此不进行详细阐述了。

但需要注意有的类虽然其析构函数是 public non-virtual[1]，但天生是为继承而生的，它唯一特别的是不具有多态性，这样的类不属于此条例。虽然它具有 public non-virtual destructor，但它就是为继承而生的。例如，std::input_iterator_tag, std::unary_function 等标准库中的类。可惜的是，这条规则往往没有得到大家的认识，在面对这样的问题时，大家习惯了为这样的类定义个 public non-virtual destructor，请建议不要那么做了。

规则 6.2.3 *考虑使用 virtual 函数声明非 public，而将 public 函数声明非 virtual*

此方法常被称为 NVI(non-virtual interface)，它是实现 Template Method Pattern 独特的一种表现形式。Template Method 在父类中以一个 public 接口实现算法的骨架，并以 private virtual 挂接许多回调钩子供子类改写。

规则 6.2.4 *区分 pure virtual, virtual, non-virtual 函数的语义*

pure virtual 是为了让子类只继承其接口；virtual 是让子类继承接口和一份默认的实现；non-virtual 是让子类继承其接口和一份强制性的实现。

当定义一个类时，清晰、准确的使用这三个特性，有助于别人理解类设计的意图。

规则 6.2.5 *改写 virtual 函数时，提供显式的 override 标示*

在子类中改写虚函数时，提供显式的 override 将改善其 API 的可读性，并增强了编译器的安全性检查。

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

INTERFACE(BbCfgListener)
{
    virtual Status onRbConfiging() = 0;
    virtual void onRbCfgFailed() = 0;
};
```

[1]此类为继承而生，但不具有多态性的类应该定义一个 protected non-vritual destructor。

```
#include "bb/role/BbCfgListener.h"

struct BbState : BbCfgListener
{
    BbState();

    Status confirm();
    Status rollback();
    void   release();

private:
    Status onRbConfiging() override;
    void   onRbCfgFailed() override;

private:
    enum{ IDLE, ACTIVE, RECFG } state;
};
```

但可惜 override 关键字直到 C++ 11 才被列入标准，此时提供类似的机制是一件值得做的工作。

```
#define OVERRIDE virtual
```

只不过此处的 OVERRIDE 需要在函数头部进行声明，当然也缺乏了编译器的安全性检查。它的存在仅仅是为了改善 API 的意图。

```
#include "bb/role/BbCfgListener.h"

struct BbState : BbCfgListener
{
    BbState();

    Status confirm();
    Status rollback();
    void   release();

private:
    OVERRIDE Status onRbConfiging();
    OVERRIDE void   onRbCfgFailed();

private:
    enum{ IDLE, ACTIVE, RECFG } state;
};
```

规则 6.2.6 *为多态基类声明为* virtual destructor

何时需要给类增加 virtual destructor 常常让人抓狂。记住两点规则：

1. As base class
2. Polymorphic

这两个条件必须同时满足，才为类增加 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，为继承而生，但没有多态的语义时，Herb Sutter, Andrei Alexandrescu 建议显式地定义 protected non-vritual destructor。可惜标准库并没有这么做，例如 std::input_iterator_tag, std::unary_function 等标准库中的类。

但这不妨碍我们这样去设计，以便透露出本类只可继承，并无多态的设计意图。

规则 6.2.7 禁止在 constructor, destructor 中调用虚函数

在 constructor, destructor 期间，子类对象尚未构造，或已经销毁，所以在其父类的 constructor, destructor 中调用虚函数未能像我们期望的那样发生多态行为，所以禁止在 constructor, destructor 中调用虚函数。

规则 6.2.8 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，所以被建议不适用，以免产生反直觉的行为。

规则 6.2.9 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 6.2.10 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，常常使用函数对象来实现。

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 Functions and Operators

7.1 Function

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。长函数往往与复杂的逻辑判断，嵌套过深等坏味道，理解它们的业务规则是一件痛苦的事情。

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。

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

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 最重要的就是梳理出主干，并遵守如下 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

```
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);
   }
}
```

虽然本例以 Java 为例，但重点没有偏离，规则同样适用于 C/C++。

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。

规则 7.1.5 将局部变量的作用域最小化

这条规则可能对 C 程序员更有指导意义，但改变这样的陋习可能需要一段时间。例如局部于 for 的循环变量，当它的使命完成之后，强制让编译器将其销毁，避免这个变量再次被使用而出错的风险。

```
map<string, string> address_book;
for ( auto address_entry : address_book )
{
    cout << address_entry.first << "," << address_entry.second << endl;
}
```

同样的规则也使用与其他常见，例如 if 语句，简化空指针的判断，简洁有效。

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

但也有人对于 if 语句这样的用法持反对态度，因为他们认为这样做更容易出错。

规则 7.1.6 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。一个查询函数，应该是没有副作用的；如果将修改对象状态的函数和查询函数混合，将破坏查询函数的这个有用的性质。

规则 7.1.7 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 7.1.8 对于参数类型，返回值类型，优先使用接口类型

这是基于接口编程的良好习惯，是优秀设计的体现。

规则 7.1.9 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，即带 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 7.1.10 永远不要导出具有相同参数数目的的重载方法

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。

[1]参考 Martin Flower 的著作《重构，改善既有代码的设计》

```
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 7.1.11 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。

7.2 Operators

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

转换运算符是一种危险的操作，它常常无声无息地让人犯错。谨慎使用，并非绝不使用，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

这是提升效率的一种举措，提倡使用前置的操作运算符。尤其在操作重载了 ++operator 的类对象，例如迭代器，更应该使用 ++operator。

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、人性化操作接口。

以扩展了的 placement new 为例，讲解 operator*, operator-> 的使用。

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

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

Placement 本质上就是一块内存，它是在原生内存上提供了一层直观的，人性化的操作接口。

众所周知，当定义一个对象的数组时，该对象的类必须提供一个默认构造函数。

[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。

如下例，ReleasingRbList 如果定义了一个 ReleasingRb，则此时因为 ReleasingRb 没有提供对应的默认构造函数而出现编译错误。

Placement 能否发挥其优势，是解决此类问题的灵丹妙药；在需要创建对象的时候，再 placement new 出来，这也是一种典型的延迟初始化技术。

```
#include "base/Status.h"
#include "erab/ErabId.h"
#include "rb/RbId.h"
#include "base/Placement.h"

struct ReleasingRb
{
    ReleasingRb(const ErabId& erabId, const RbId& rbId);

    const ErabId& getErabId() const;
    const RbId& getRbId() const;

private:
    ErabId erabId;
    RbId   rbId;
};

struct ReleasingRbList
{
    ReleasingRbList();

    bool contains(const ErabId& erabId) const;
    Status addReleasingRb(const ErabId& erabId, const RbId& rbId);

private:
    enum { MAX_UE_ERAB_NUM = 8 };

    BYTE n;
    Placement<ReleasingRb> relErabs[MAX_UE_ERAB_NUM];
};
```

```
#include "erab/ReleasingRbList.h"
#include "base/Assertions.h"

ReleasingRbList::ReleasingRbList() : n(0)
{
}

Status ReleasingRbList::addReleasingRb(const ErabId& erabId, const RbId& \
  rbId)
{
    ASSERT_FALSE(contains(erabId));
    ASSERT_TURE(n < MAX_UE_ERAB_NUM);

    new (relErabs[n++].alloc()) ReleasingRb(erabId, rbId);

    return DCM_SUCCESS;
}

bool ReleasingRbList::contains(const ErabId& erabId) const
{
    for (BYTE i = 0; i < n; ++i)
    {
        if(relErabs[i]->getErabId() == erabId)
        {
            return true;
        }
    }
```

```
        return false;
    }
```

relErabs[i]->getErabId() 调用时，实际上调用了 Placement::operator->，但从实现上看，给用户调用提供了很大的灵活性和方便性，这还是得倚仗了 operator-> 带来的间接性的功劳。

This page is intentionally left blank.

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 Clean Test

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》。

8.2 Clean Test

规则 8.2.1 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，因为它没有固定的规范，也不能清晰表达 TDD 的意图，更像在做一个测试。

规则 8.2.2 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，应避而远之。

规则 8.2.3 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 8.2.4 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 8.2.5 设计面向特定领域的测试语言是值得的