Contents

[Philips Architecture Curriculum 3](#_Toc150005035)

[编程思想 3](#_Toc150005036)

[设计模式 4](#_Toc150005037)

[创建型模式与对象的创建有关（Factory Method, Builder, Singleton, Abstract Factory） 6](#_Toc150005038)

[结构型模式处理类或对象的组合 (Adapter, Proxy, Façade, Bridge, Composite, Decorator) 10](#_Toc150005039)

[行为型模式对类或对象怎样交互和怎样分配职责进行描述 16](#_Toc150005040)

[架构模式 26](#_Toc150005041)

[Back-end迁移到新架构 --Strangler Fig pattern 绞杀者模式 26](#_Toc150005042)

[Layer分层 27](#_Toc150005043)

[Pipeline/Filters 27](#_Toc150005044)

[MVC(Model-View-Control) & MVVM (Model-View-ViewModel) 27](#_Toc150005045)

[微服务 30](#_Toc150005046)

[演化式架构师 32](#_Toc150005047)

[如何建模服务 33](#_Toc150005048)

[集成 33](#_Toc150005049)

[分解单块系统 37](#_Toc150005050)

[部署 39](#_Toc150005051)

[云服务模式 42](#_Toc150005052)

[DevOps&Tools 48](#_Toc150005053)

[<企业集成模式：设计、构建及部署消息传递解决方案> 49](#_Toc150005054)

[<企业应用架构模式> 50](#_Toc150005055)

[Heroku 50](#_Toc150005056)

[Pivotal Web Services 51](#_Toc150005057)

[Monolithic vs micro-service 53](#_Toc150005058)

[monolithic application 53](#_Toc150005059)

[micro-services 53](#_Toc150005060)

[Eureka = Netflix Service Discovery Server + Client 53](#_Toc150005061)

[Communicate between front-end and back-end? 54](#_Toc150005062)

[分布式应用系统的架构演变 54](#_Toc150005063)

[Authentication and Authorization 57](#_Toc150005064)

[AD LDS Server 57](#_Toc150005065)

[邮件发送 59](#_Toc150005066)

[Domain-specific language 59](#_Toc150005067)

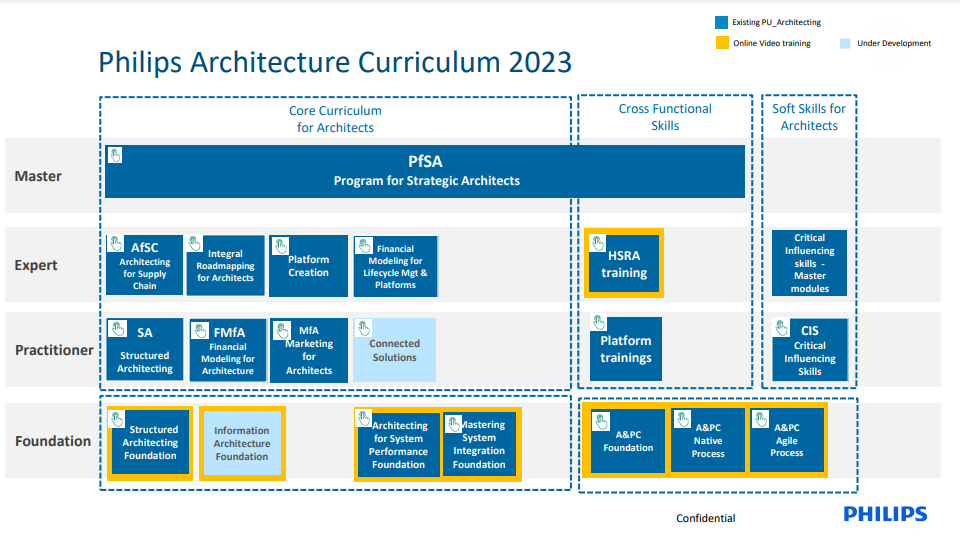
[Drools 59](#_Toc150005068)

[测试 61](#_Toc150005069)

[压力测试 61](#_Toc150005070)

[Office Note 61](#_Toc150005071)

# Philips Architecture Curriculum



# 编程思想

Books:《Unix编程艺术》

UNIX原本是为技术人员设计的操作系统，同时也表明设计的信念：最终用户永远比操作系统设计人员更清楚他们究竟需要什么

C语言本身就是Unix的一项发明，而ANSI C程序库表达了相当大一部分的Unix服务

Unix哲学：一个程序只做一件事，并做好。程序要协作。程序要能处理文本流，因为这是最通用的接口。编程的核心是数据结构，而不是算法。复杂的算法比简单算法更容易出bug, 更难实现，尽量使用简单的算法配合简单的数据结构

**模块原则**：使用简洁的接口拼命简单的部件

要编制复杂软件而又不至于一败涂地的唯一方法就是降低其整体复杂度------用清晰的接口把若干简单的模块组合成一个复杂软件。

**组合原则**：设计时考虑拼接组合

在经典的Unix下，多数程序都尽可能采用简单过滤器的形式，即将一个输入的简单文本流处理为一个简单的文本流输出。（Unix中，文本流之于工具，就如同在面向对象环境中的消息之于对象）

**分离原则**：策略同机制分离，接口同引擎分离

使X成为一个通用图形引擎，而将用户界面风格留给工具包或者系统的其它层次来决定。

将应用按照一个库来编写，这个库包含许多C服务程序，而整个应用的控制流程则用脚本来撰写

将应用程序分成可以协作的前端和后端进程，通过套接字上层的专用应用协议进行通讯。前端实现策略，后端实现机制。

**表示原则**：把知识叠入数据以求逻辑质朴而健壮

数据要比编程逻辑更容易驾驭。在设计中，你应该主动将代码的复杂度转移到数据之中去。比如用一个数组初始化器来表示转换表，比switch语句更清楚

**生成原则**：避免手工hack,尽量编写程序去生成程序。

程序中的任何手工hacking都是滋生错误的温床。由程序生成代码几乎（在各个层次）总是比手写代码廉价并且更值得依赖

**优化原则**：先求运行，再求正确，最后求快

先给你的设计做个未优化的、运行缓慢、很耗内存但是正确的实现，然后进行系统地调整，寻找那些可以通过牺牲最小的局部简洁性而获得较大性能提升的地方。

# 设计模式

对使用传统语言的程序设计专家们的研究表明，**其知识和经验并非是简单地围绕语法来组织的，而是围绕着诸如算法、数据结构、习惯用语，设计模式，架构模式等更大的概念结构来组织**

如何把当前的设计问题与已知的计划、算法、数据结构，习惯用语等进行匹配

软件生命周期：1）原型阶段，快速建立原型，然后进行增量式的修改，直至满足一组基本需要。此时软件中的类层次通常直接反映了原始问题域中的各个实体。该阶段主要的复用方式得到通过继承进行白箱复用。2) 扩展阶段，软件需要满足更多的需求，软件类层次不再与任何问题域匹配，而是多个问题域的混合反映，并且类中定义了许多不相关的操作和实例变量。3）重构阶段。将类拆分为专用和通用的构件，把各个操作在类层次上提或下放到合适的类中，并使各个类的接口合理化。这个阶段会产生许多新类型的对象，它们通常是通过分解而不是继承原有的对象而得到的，因此黑箱复用代替了白箱复用。最终使软件满足更多需求和达到更高复用性。

客户请求是使对象执行操作的唯一方法，操作又是对象改变内部数据的唯一方法。

面向对象设计最困难的部分是将系统分解成对象集合。因为要考虑许多因素：封装、粒度、依赖关系、灵活性、性能、演化、复用等等。

动态绑定允许你在运行时刻彼此替换有相同接口的对象。这种可替换性就称为多态

**类继承与接口继承：**

类继承根据一个对象的实现定义了另一个对象的实现。它是代码和表示的共享机制。接口继承（或子类型化）描述了一个对象什么时候能被用来替代另一个对象。

C++中接口继承的标准方法是公有继承一个含纯虚成员函数的类。C++中纯接口继承接近于公有继承纯抽象类，纯实现继承或纯类继承接近于私有继承。

1. 针对接口编程，而不是针对实现编程

**类继承和对象组合：**

类继承通过生成子类的复用，也称为白箱复用。在继承方式中，父类的内部细节对子类可见。

对象组合实现更复杂的功能，也称为黑箱复用。因为对象的内部细节是不可见，对象只以“黑箱“的形式出现。

类继承缺点：1）编译时刻静态定义，所以无法在运行时刻改变从父类继承的实现。2）继承常被认为“破坏了封装性“，因为继承对子类揭示了其父类的实现细节。3）子类中的实现与它的父类 紧密依赖，以至于父类实现中的任何变化必然会导致子类发生变化

对象组合是通过获得对其他对象的引用而在运行时刻动态定义的，**组合要求对象遵守彼此的接口约定，**进而要求更仔细地定义接口。对象的实现基于接口写的，所以实现存在较少的依赖关系。优化使用对象组合有助于你保持每个类被封装，并被集中在单个任务上。这些类和类继承层次会保持较小规模。

1. 优先使用对象组合，而不是类继承

根据经验表明：设计者往往过度使用了继承这种复用技术。但依赖于对象组合技术的设计却有更好的复用性

委托是一种组合方法，便于运行时刻组合对象操作以及改变这些操作的组合方式。

聚合意味着一个对象拥有另一个对象或对另一个对象负责。意味着聚合对象和其所有者具有相同的生命周期。

关联（引用）关系意味着一个对象仅仅知道另一个对象。关联对象可能请求彼此的操作，但是它们不为对方负责。**关联是一种比聚合要弱的关系，它只标识了对象间较松散的耦合关系。**

C++中，聚合可以通过定义表示真正实例的成员变量来实现，但更通常的是将这些成员变量定义为实例指针或引用；关联也是以指针或引用来实现

从根本上来讲，是聚合还是关联是由你的意图而不是由显式的语言机制决定的。聚合关系使用较少且比引用关系更持久；而引用关系则出现频率较高，但有时只存在于一个操作期间，引用也更具动态性，使得它在源代码中更难被辨别出来。

组合面向对象系统中行为的三种方法：**类继承，对象组合和参数化类型（如模板）。**对象组合允许你在运行时刻改变被组合的行为，但是它存在间接性，比较低效；继承允许你提供操作的缺省实现，并通过子类重定义这些操作；参数化类型允许你改变类所用到的类型。但是继承和参数化类型都不能在运行时刻改变。

动态的、高度参数化的软件比静态软件更难于理解。

通常你通过引入额外的间接层次获得灵活性和可变化的同时，你也使设计变得更复杂并牺牲了一定的性能。

设计模式是针对特定问题的简洁而优雅的解决方案。

单继承和多重继承机制用于基于类的模式；对象组合机制用于对象式模式。类模式处理类和子类之间的关系，这些关系通过继承建立，是静态的，在编译时刻便确定下来；对象模式处理对象间的关系，这些关系在运行时刻是可以变化的，更具动态性。

创建型类模式将对象的部分创建工作延迟到子类，而创建型对象模式则将它延迟到另一个对象中。

结构型类模式使用继承机制来组合类，而结构型对象模式则描述了对象的组装方式。

行为型类模式使用继承描述算法和控制流，而行为型对象模式则描述一组对象怎样协作完成单个对象所无法完成的任务

**Factory method**（Virtual constructor虚构造器）：定义一个用于创建对象的接口，让子类决定将哪一个类实例化

**Builder**：将复杂对象构建与表示分离，使得同样的构建过程可以创建不同的表示

**Singleton**：保证一个类仅有一个实例，并提供全局访问点

**Abstract Factory**：创建相互依赖对象的接口，而无需指定它们具体的类

**Adapter**（包装器Wrapper）：解决接口不兼容，时常还要负责提供那些被匹配的类所没有提供的功能

**Proxy**： 控制对象的访问。

**Facade**： 简化接口：为子系统中的一组接口提供一个一致的界面，Façade模式定义了一个高层接口，使得子系统更加容易使用

**Bridge**： 将抽象部分与实现部分分离，使它们可以独立变化

**Composite**：将对象组合成树形结构以表示“部分-整体”的层次结构，使用户对单个对象和组合对象的使用具有一致性

**Decorator** （包装器Wrapper）：动态地给对象添加额外的职责

**Template Method**：定义算法骨架，而将一些步骤延迟到子类中

**Observer**（发布-订阅publish-subscribe）：定义对象间的一对多依赖关系，当一个对象的状态发生改变时，所有依赖它的对象都得到通知并被自动更新

**Strategy**（策略Policy）：将算法封装在对象中，使它们可相互替换

**State**（状态对象）：封装一个对象的状态，允许一个对象在其内部状态改变时改变它的行为。

**Mediator**：用一个中介对象Mediator来封装一系列的对象交互

**Chain of Responsibility** ：为解除请求的发送者和接收者之间耦合，而使多个对象都有机会处理这个请求。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它

**Command** ：将请求封装在对象中，这样它可作为参数来传递，也可以被存储在历史列表里，或者以其他方式使用。

## 创建型模式与对象的创建有关（Factory Method, Builder, Singleton, Abstract Factory）

抽象了实例化过程，客户仅与抽象类定义的接口交互，而不使用具体类的接口。

什么时候，在哪里，谁创建，怎样创建？配置可以是静态的（即在编译时指定），也可以是动态的（在运行时）

通常设计以使用Factory Method开始，并且当设计者发现需要更大的灵活灵活性时，设计便便会向其他创建型型模式演化

* **Factory Method (工厂方法) - 对象创建模式 （将实例化委托给另一个对象）**

Virtual constructor虚构造器, FactoryMethod()使一个类的实例化延迟到其子类

A diagram of a process

Description automatically generated with low confidence

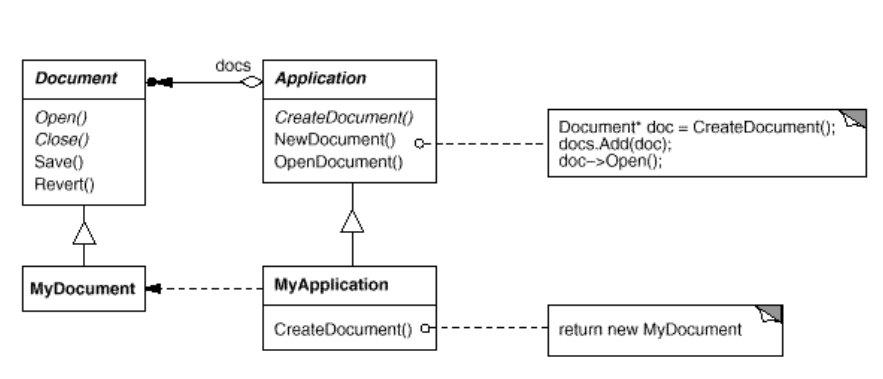
**应用：**

1. 工厂方法给子类一个hook以提供对象的扩展版本

框架中，框架使用抽象类维护对象间关系，如向用户显示多个文档这个框架，两个主要的抽象类是Application (等价于Creator) and Document (等价于Product). 客户必须通过它们的子类来做与具体应用相关的实现。

工厂方法不再将与特定应用有关的类绑定到你的代码中，代码仅处理Document接口

用工厂方法在一个类的内部创建对象通常比直接创建对象更灵活。

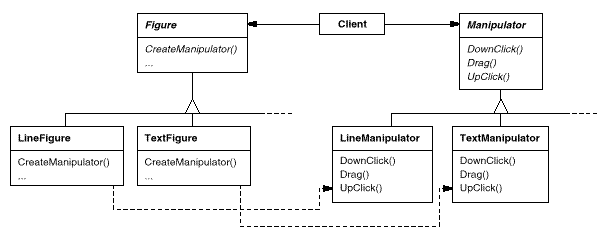


CreateDocument()是一个工厂方法，因为它负责“生产”一个对象

Application依赖于它的子类来定义工厂方法，返回一个Document实例

1. 实现平行的类层次

当一个类将它的一些职责委托给一个独立类的时候，就产生了平行类层次。如图形交互，通常需要存储和更新在给定时刻记录操纵状态的信息，而这个状态仅仅在操纵时需要，不建议被保存在图形对象中。最好使用一个独立的Manipulator (等价于Product)对象实现交互并保存所需要的任何与特定操纵相关的状态。当用户操纵图形时，不同的图形(等价于Application)有不同的行为，使用不同的Manipulator子类来处理特定的交互。



**使用模板以避免创建子类**

class Creator {

public:

virtual Product\* CreateProduct() = 0; // C++工厂方法一般是虚函数且常常是纯虚函数

};

template <class TheProduct>

Product\* StandardCreator: public Creator {

public:

virtual Product \* CreateProduct();

};

template <class TheProduct>

Product\* StandardCreator<TheProduct>::CreateProduct() {

return new TheProduct;

}

客户仅提供产品类，而不需要创建Creator的子类

class MyProduct : public Product {

public:

MyProduct();

};

StandardCreator<MyProduct> myCreator;

参数化工厂方法

一个Creator类，该类拥有一个以类标识符为参数的工厂方法Creat。类标识符指定要被实例化的类。当一个对象存盘时，它首先写类标识符，然后是它的实例变量。当它从磁盘中重构该该对象时，它首先读取的是类标识符。

一但类标识符被读取后，这个框架就将该标识符作为参数，调用Create(), 从而到构造器中查询相应的类并用它实例化对象。最后Create()调用对象的Read(), 读取磁盘上剩余的信息并初始化该对象的实例变量。

class Creator {

public:

virtual Product\* Create(ProductId) ;

};

Product \* Creator::Create(ProductId id) {

if (id == MINE) return new MyProduct;

if (id == YOURS) return new YourProduct;

…

return 0;

}

Product \* MyCreator::Create(ProductId id) {

if (id == THEIRS) return new TheirProduct;

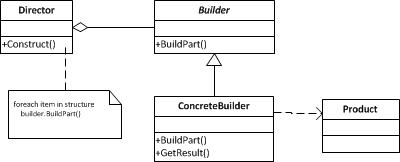
…

return Creator::Create(id);

}

* **Builder生成器 - 对象创建模式**

Builder是在导向者的控制下一步一步构造产品的；通过封装一个复杂对象的创建和表示方式，提高了对象的模块性

 A picture containing text, diagram, screenshot, line

Description automatically generated

客户创建Director对象，并用ConcreateBuilder对象进行配置。ConcreateBuilder创建该产品的内部表示并定义它的装配过程。

* **Singleton单件 - 对象创建模式**

保证一个类仅有一个实例，并提供全局访问点

Class Singleton {

public:

static Singleton\* Instance(); //通过类方法或静态成员函数，访问唯一实例

protected:

Singleton(); // 避免用户创建实例

private:

static Singleton\* \_instance;

};

Singleton\* Singleton::\_instance = nullptr;

Singleton\* Singleton::Instance() {

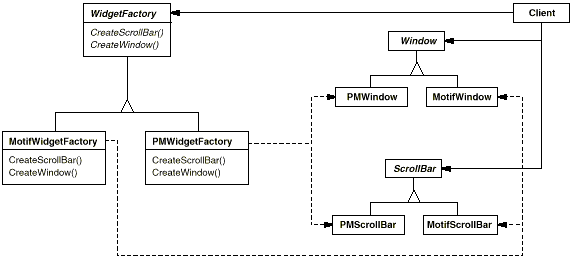
if (\_instance == nullptr) \_instance = new Singleton; //使用Lazy初始化

return \_instance;

}

* **Abstract Factory （抽象工厂） - 对象创建模式**

创建相互依赖对象的接口，而无需指定它们具体的类。为保证界面风格标准间的可移植性，一个应用不应该为一个特定的界面外观硬编码它的窗口组件。在整个应用中实例化特定界面风格的窗口组件类将使得以后很难改变界面风格



客户端仅使用由WidgetFactory(抽象工厂), Window and ScrollBar(抽象产品)

抽象工厂通常为每种产品定义一个操作，如CreateWindow(); 增加一种新的产品要求改变抽象工厂的接口

具体工厂MotifWidgetFactory创建Motif风格的窗口和滚动条

一个应用一般每个产品系列中只需一个具体工厂（通常实现为一个Singleton），**通常在编译时刻配置或在运行时刻创建一个具体工厂**，即你只需改变具体工厂即可使用不同的产品配置，这是因为一个抽象工厂创建了一个完整的产品系列。

注意：当一个系列中的产品对象被设计成一起工作时，一个应用一次只能使用同一个系列中的对象。难以支持新种类的产品，这是因为抽象工厂接口确定了可以被创建的产品集合。

## 结构型模式处理类或对象的组合 (Adapter, Proxy, Façade, Bridge, Composite, Decorator)

结构型类模式采用继承机制来组合接口或实现

结构型对象模式描述如何对一些对象进行组合

Adapter模式主要是为了解决两个已有接口之间不匹配的问题，不考虑这些接口是怎样实现的，也不考虑它们各自可能会如何演化。

Bridge模式则对抽象接口与它的多个实现部分进行桥接

Façade模式不仅仅是另外一组对象的适配器，Façade也定义一个新的接口，而Adapter使两个已有的接口协同工作，而不是定义一个全新的接口

Decorator旨在使你能够不需要生成子类即可给对象添加职责，避免了静态实现所有功能组合，从而导致子类急剧增加。

Composite使多个相关的对象能够以统一的方式处理，多重对象可以当作一个对象来处理

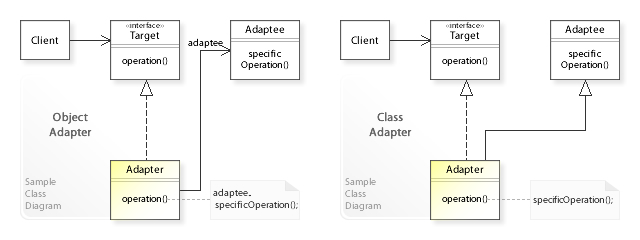
* **Adapter适配器 （包装器Wrapper） - 类结构模式，对象结构模式**

解决接口不兼容，时常还要负责提供那些被匹配的类所没有提供的功能

Adapter的工作量取决于Target接口接口与Adaptee接口的相似程度

类版本：继承接口类和私有继承实现类

对象版本：将实现类的实例作为接口类的组合部分



* **Proxy -对象结构型模式**

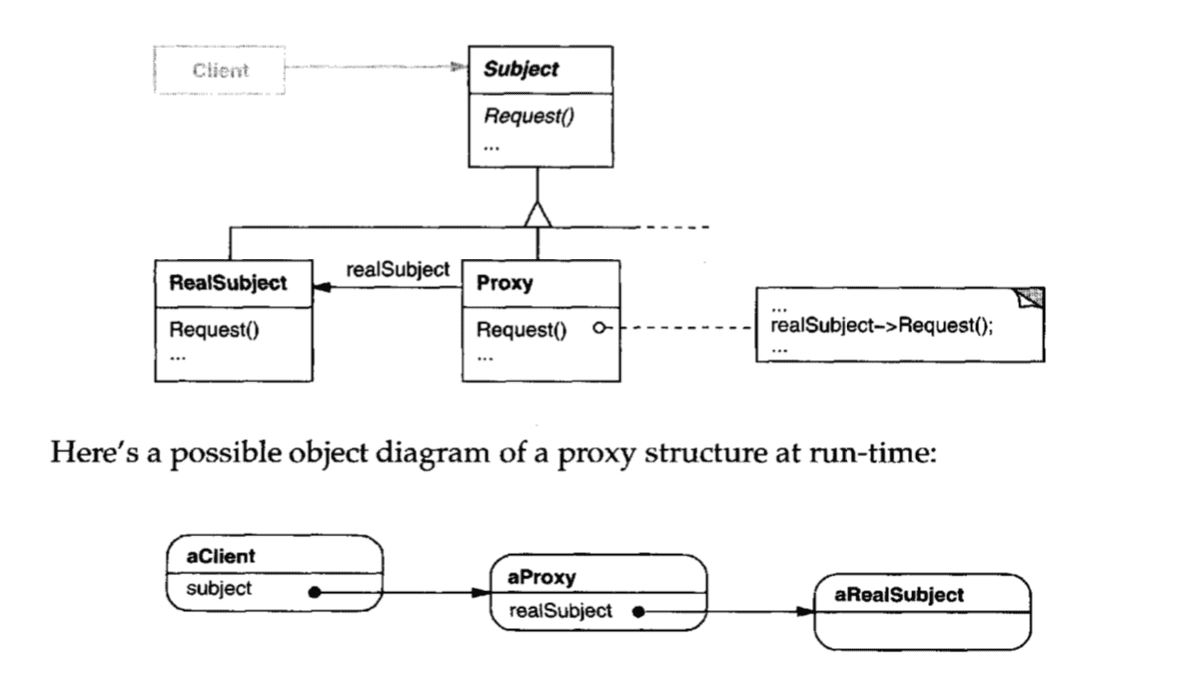
控制对象的访问。

远程代理Remote Proxy：为一个对象在不同地址空间提供局部代表

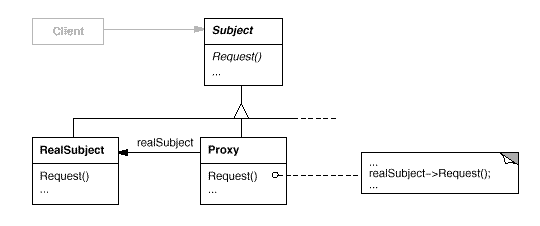
虚代理Virtual Proxy：根据需要创建开销很大的对象

保护代理Protection Proxy：控制对原始对象的访问

智能指引Smart Reference: 取代简单指针，在访问对象时执行一些附加操作



提供相同的接口，代理用来替代实体



注意区别：

Adapter为适配对象提供不同接口；Proxy提供相同接口；Decorator为对象添加功能。

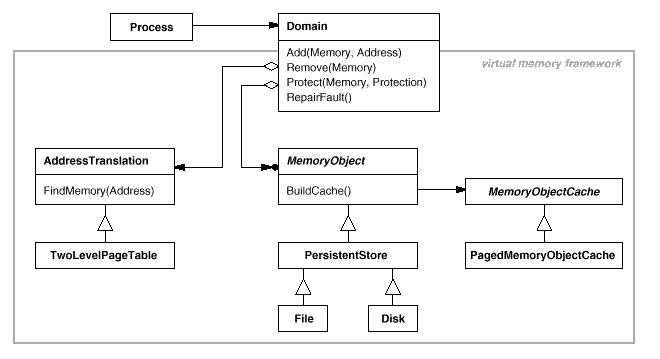
代理实现与Decorator实现类似

* **Facade -对象结构型模式**

简化接口：为子系统中的一组接口提供一个一致的界面，Façade模式定义了一个高层接口，使得子系统更加容易使用

降低编译依赖性：在大型软件系统中降低编译依赖性至关重要。在子系统类改变时，希望尽量减少重编译工作以节省时间，用Façade可以降低编译依赖性，限制重要系统中较小的变化所需的重编译工作

通常来讲，仅需要一个Façade对象，使用Singleton模式



虚拟存储框架将Domain作为其façade，一个Domain代表一个地址空间。它提供了虚存地址到内存对象、文件系统或后备存储设备的偏移量之间的一个映射, …

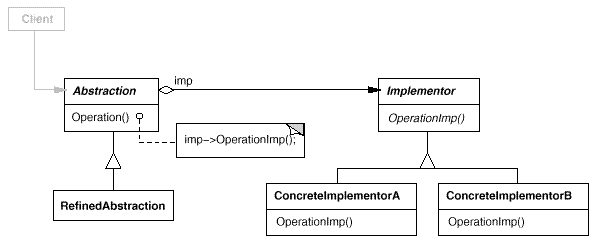
* **Bridge桥接 - 对象结构型模式**

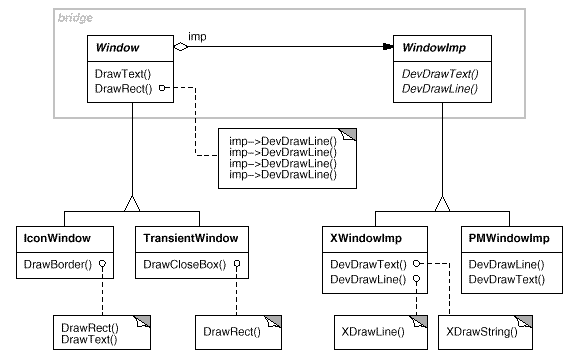
继承机制将抽象部分与它的实现部分固定在一起，使得难以对抽象部分和实现部分独立地进行修改、扩充和重用

将抽象部分与实现部分分离，使它们可以独立变化，有助于降低对实现部分编译时刻的依赖性，当改变一个实现类时，并不需要重新编译抽象类和它的客户程序。

类的抽象以及它的实现都可以通过生成子类的方法加以扩充。抽象类Abstraction和实现类Implementor之间的关系称为桥接。

一般来讲，Implementor接口仅提供基本操作，而Abstraction则定义了基于这些基本操作的较高层次的操作。





将Window抽象和它的实现部分分别放在独立的类层次结构中。其中一个类层次结构针对窗口接口(Window, IconWindow, TransientWindow)，另外一个独立的类层次结构针对平台相关的窗口实现部分，对Window子类的所有操作都是用WindowImp接口中的抽象操作实现的。这就将窗口的抽象和系统平台相关的实现部分分离出来。

* **Composite 组合 -对象结构型模式**

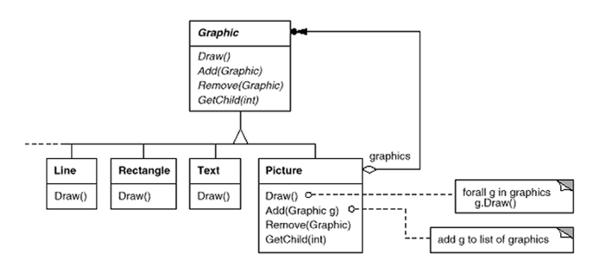
在绘图编辑器中，用户可以使用简单的组件创建复杂的图表。用户可以组合多个简单组件以形成一些较大的组件。一个简单的实现方法是为Text and Line这样的图元定义类，另外定义含这些图元的容器类Container，然而这种方法存在一个问题：使用这些类的代码必须区别对待图元对象与容器对象。区别使用会使得程序更加复杂。

Composite使用户对单个对象和组合对象的使用具有一致性，使得用户不必对这些类进行区别。通常用户不知道（也不关心）处理的是一个叶节点还是一个组合组件，为了达到这个目的，Composite类应为Leaf and Composite类尽可能多定义一些公共操作。Composite类通常为这些操作提供缺省的实现，而Leaf and Composite子类可以对它们进行重定义

A diagram of a component

Description automatically generated with medium confidence

用户使用Component类接口与组合结构中的对象进行交互。如果接收者是一个叶节点，则直接处理请求；如果接收者是Composite，它通常将请示发送给它的子部件，在转发请求之前或之后可能执行一些辅助操作。



Composite模式的关键是一个抽象类Component，即可以代表图元，又可以代表图元的容器

class Composite;

class Component {

public:

virtual Composite\* GetComposite() { return nullptr;}

};

class Composite : public Component {

public:

virtual Composite\* GetComposite() { return this; }

void Add(Component\*);

…

private:

List<Component\*> \_components;

};

客户端

void fun(Component\* component) {

If (component->GetComposite()) component->Add(new Leaf);

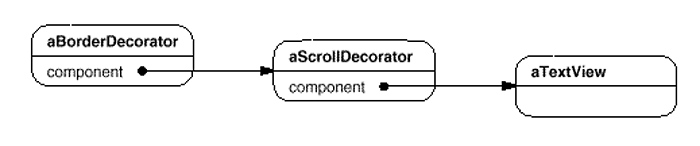
}

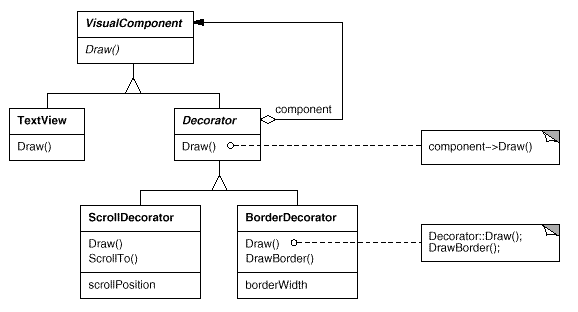
* **Decorator （包装器Wrapper） -对象结构型模式**

动态地给对象添加额外的职责

使用继承添加功能，是静态的，用户不能控制添加功能的方式和时机

将组件嵌入另一个对象中，由这个对象添加功能，我们称这个嵌入的对象为装饰，装饰与被装饰接口一致，因此它对使用该组件的客户透明。





class VisualComponent {

public:

visualComponent();

virtual void Draw();

};

// Decorator维持一个指向Component对象的指针，并定义一个与Component接口一致的接口

class Decorator : public VisualComponent {

public:

Decorator(VisualComponent\*);

virtual void Draw();

private:

VisualComponent\* \_component;

};

void Decorator::Draw() {

\_component->Draw();

}

// Decorator子类定义特殊的装饰功能

class BorderDecorator : public Decorator {

public:

BorderDecorator(VisualComponent\*, int borderWidth);

virtual void Draw();

};

void BorderDecorator::Draw(){

Decorator::Draw();

DrawBorder(\_width); //绘制边框

}

客户端

void Window::SetContents(VisualComponent\* contents) {

//…

}

window->SetContents(new TextView);

// 用户在运行时刻动态增加功能

window->SetContents(new BorderDecorator(new ScrollDecorator(new TextView), 1));

## 行为型模式对类或对象怎样交互和怎样分配职责进行描述

封装变化是很多行为模式的主题

模式通常定义一个抽象类来描述这些封装变化的对象，并且通常该模式依据这个对象来命名，例如：

一个Strategy对象封装一个算法

一个State对象封装一个与状态相关的行为

一个Mediator对象封装对象间的协议，为各Colleague对象间的请求提供路由并集中通信

一个Iterator对象封装访问和遍历一个聚集对象中的各个构件的方法

Observer模式通过引Observer and Subject对象来分布通信，通过定义一个接口来通知目标中发生的改变，从而将发送者（目标）与接收者（观察者）解耦

Command对象提供一个提交请求的简单接口（如Execute()），将发送者和接收者之间的连接定义在一个单独的对象使得该发送者可以与不同的接收者一起工作，从而发送者与接收者解耦。

职责链模式通过沿一个潜在接收者链传递请求而将发送者与接收者解耦

如果不使用这些模式的话，通常这些新对象的功能就会变成这些已有对象的难以分割的一部分。例如一个Strategy的代码可能会被嵌入到其Context类中，而一个State的代码可能会在该状态的Context类中直接实现

当合作的对象直接互相引用时，它们变得互相依赖，这可能会对一个系统的分层和重用性产生负页影响。命令、观察者、中介者和职责链等模式都涉及如何对发送者和接收者解耦

控制流转移 -> 对象间的关系

行为类模式使用继承机制在类间分派行为

行为对象模式使用对象复合而不是继承

* **Template Method -类行为型模式**

模板方法是一个算法的抽象定义，它逐步地定义该算法，每一步调用一个抽象操作或一个原语操作，子类定义抽象操作以具体实现该算法

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中

一个模板方法用一些抽象的操作定义一个算法，而子类将重定义这些操作以提供具体的行为。

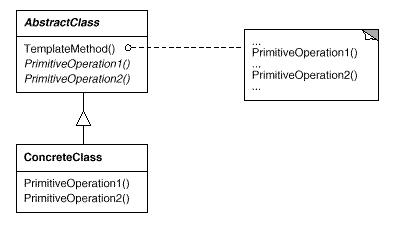
通过使用抽象操作定义一个算法中的一些步骤，模板方法确定了它们的先后顺序，但它允许子类改变这些具体步骤以满足它们各自的需求。

一次性实现一个算法的不变部分，并将可变的行为留给子类来实现

模板方法导致一种反向的控制结构，这种结构有时被称为“好莱坞法则”即“别找我们，我们找你”。这指的是一个父类调用一个子类的操作，而不是相反。

钩子操作hook operations, 它提供了缺省的行为，子类可以在必要时进行扩展，一个钩子操作在缺省操作通常是一个空操作，子类可以通过重定义父类的操作来扩展该操作的行为，其间可显式地调用父类操作

模板方法使用继承来改变算法的一部分。Strategy使用委托来改变整个算法。



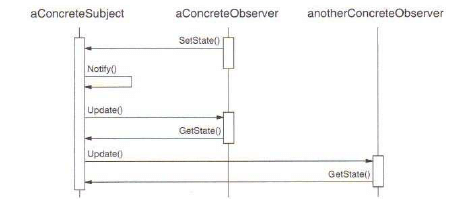
* **Observer模式 （发布-订阅publish-subscribe） -对象行为型模式**

定义对象间的一对多依赖关系，当一个对象的状态发生改变时，所有依赖它的对象都得到通知并被自动更新

典型的Observer例子是MVC，模型状态发生变化，模型的所有视图都会得到通知

A picture containing text, diagram, plan, technical drawing

Description automatically generated



注意发出改变请求的Observer对象并不立即更新，而是将其推迟到它从目标得到一个通知之后。Notify不总是由目标对象调用，它也可被一个观察者或其它对象调用。

目标和它的观察者依赖于通知机制来保持一致。但到底是哪个对象调用Notify来触发更新？

由目标对象的状态设定操作在改变目标对象的状态后自动调用Notify,缺点是多个连续操作会产生多次连续的更新，可能效率较低

让客户负责在适当的时候调用Notify.优点是客户在一系列的状态改变完成后再一次性触发更新，避免了不必要的中间更新。

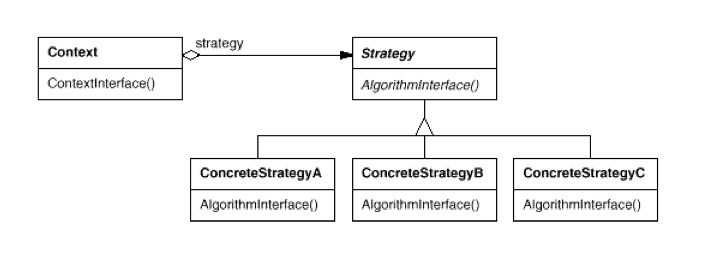
Observer模式的实现经常需要让目标广播关于其改变的其他一些信息。目标将这些信息作为Update操作一个参数传递出去。这些信息的量可能很小，也可能很大。一个极端情况是，目标向观察者发送关于改变的详细信息，而不管它们是否需要。我们称之为推模型push model. 另一个极端是拉模型pull model, 目标除最小通知什么也不送出，而在此之后由观察者显式地向目标询问细节。

* **Strategy 模式 （政策Policy） -对象行为型模式**

将算法封装在对象中，使它们可相互替换

继承提供简单的支持多种算法的方法，你可以直接生成一个Context类的子类，从而给它以不同的行为。但这会将行为硬编码到Context中，便算法的实现与Context的实现混合起来，导致Context难以理解，难以扩展，而且无法动态改变算法。

将算法封装在独立的Strategy类中使得你可以独立于其Context改变它，使它易于切换、易于扩展。

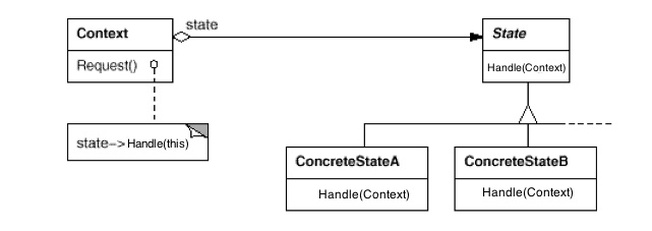


当算法被调用时，Context可以将该算法所需要的所有数据都传递给该Strategy,或者Context将自身作为一个参数传递给Strategy操作，让Strategy在需要时可以回调Context.

客户通常创建并传递一个ConcreteStrategy对象给Context, 这样客户仅与Context交互。通常有一系列的ConcreteStrategy类可供客户从中选择。

* **State 模式 （状态对象） -对象行为型模式**

封装一个对象的状态，允许一个对象在其内部状态改变时改变它的行为。



客户可用状态对象来配置一个Context, 一旦一个Context配置完毕，它的客户不再需要直接与状态对象打交道

Context or ConcreteState子类都可决定哪个状态是另外哪一个的后继者，以及是在何种条件下进行状态转换。如果状态转换规则是固定的，那么在Context中完全实现；也可以让State子类指定它们的后继状态以及何时进行转换，虽然灵活，但一个State子类拥有其他子类的信息，产生了子类之间的实现依赖

通常定义新的子类可以很容易增加新的状态和转换

决定状态转移的逻辑不在单块的if or switch语句中，而是分布在State子类之间，将每一个状态转换和动作封装到一个类中，就把着眼点从执行状态提高到整个对象的状态。

从Context角度看，状态转换是原子的，只需重新绑定一个变量（即Context的State对象变量），而无需为多个变量赋值。

基于映射表方法：对每一个状态，一张表将每一个可能的输入映射到一个后续状态。简单的方法是基于查找表，将条件代码或State模式下的虚函数映射为一个查找表。表的主要好处是它们的规则性，你可以通过更改数据而不是更改程序代码来改变状态转换的准则。

例: 交通灯转换

class Traffic {

public:

Traffic() { \_state = new Red(this) ; }

void ChangeState(State\* state) { \_state = state ; }

void Handle() { \_state->Handle(); }

private:

State\* \_state;

};

class State {

public:

virtual void Handle() = 0;

};

class Red : public State {

public:

Red(Traffic\* context);

void Handle() {

// … process when red light

// 亮红灯60秒

\_context->ChangeState(new Yellow(\_context));

}

private:

Traffic\* \_context;

};

class Yellow : public State {

public:

Yellow (Traffic\* context);

void Handle() {

// … process when Yellow light

// 亮黄灯10秒

\_context->ChangeState(new Green(\_context));

}

private:

Traffic\* \_context;

};

class Green : public State {

public:

Green (Traffic\* context);

void Handle() {

// … process when Green light

// 亮绿灯30秒

\_context->ChangeState(new Red(\_context));

}

private:

Traffic\* \_context;

};

客户端：红灯60秒 ->黄灯10秒 -> 绿灯30秒 -> …

Traffic traffic;

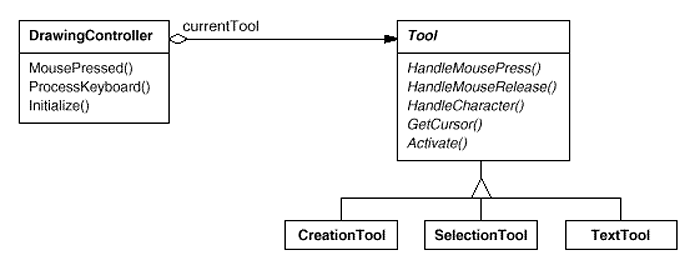
while (True) {

traffic.Handle();

}

TCPConnection对象的状态转换：连接已建立Established, 正在监听Listening, 连接已关闭Closed

交互式绘图程序：编辑器的行为随当前的工具而变，当一个绘制工具被激活时，我们创建图形对象；当选择工具被激活时，我们选择图形对象；等等



* **Mediator模式**

虽然将一个系统分割成许多对象通常可以增强可复用性，但是对象间如何相互协作？如何互相了解对方？

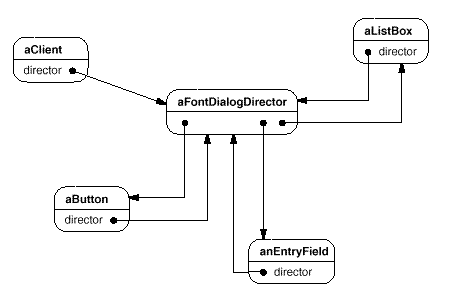
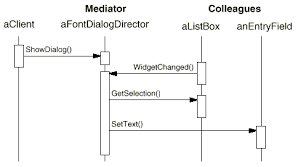
简单的方法是：对等对象可以保持显式的对对方的引用，但那会增加它们之间的耦合度。极端情况下，每一个对象都要了解所有其他的对象

用一个中介对象Mediator来封装一系列的对象交互。

中介者负责控制和协调一组对象间的交互。中介者充当一个中介以使组中的对象不再相互显式引用。**这些对象仅知道中介者**，从而减少了相互连接的数目。

用Mediator和各Colleague间的一对多的交互来代替多对多的交互

中介者模式将交互的复杂性变为中介者的复杂性。因为中介者封装了协议，它可能变得比任一个Colleague都复杂。

列表框选择联动输入域：

1. 列表框告诉它的操作者它被改变了
2. Mediator(也称为Director)从列表框中得到选中的选择项
3. Mediator将该选择项传递给入口域
4. 现在入口域已有正文，Mediator使得用于发起一个动作（如“半黑体”）的某些按钮可用

class DialogDirector {

public:

virtual void WidgetChanged(Widget\*) = 0;

};

class Widget {

public:

Widget(**DialogDirector\*);**

virtual void Changed();

virtual void HandleMouse(MouseEvent& event);

private:

**DialogDirector\* \_director;**

};

void Widget::Changed() { **\_director->WidgetChanged(this);** }

void Widget::HandleMouse(MouseEvent& event) {

//….

Changed();

}

class ListBox : public Widget {

public:

ListBox(DialogDirector\*);

virtual const char\* GetSelection();

};

class EntryField : public Widget {

public:

EntryField(DialogDirector\*);

virtual void SetText(const char\* text);

};

class FontDialogDirector : public DialogDirector {

public:

virtual void WidgetChanged(Widget\*) ;

protected:

virtual void CreateWidgets();

private:

ListBox\* \_fontList;

EntryField\* \_fontName;

};

void FontDialogDirector::CreateWidgets() {

\_fontList = new ListBox();

\_fontName = new EntryField();

//…

}

void FontDialogDirector::WidgetChanged(Widget\* theChangedWidget) {

If (theChangedWidget == \_fontList) {

\_fontName->SetText(\_fontList->GetSelection());

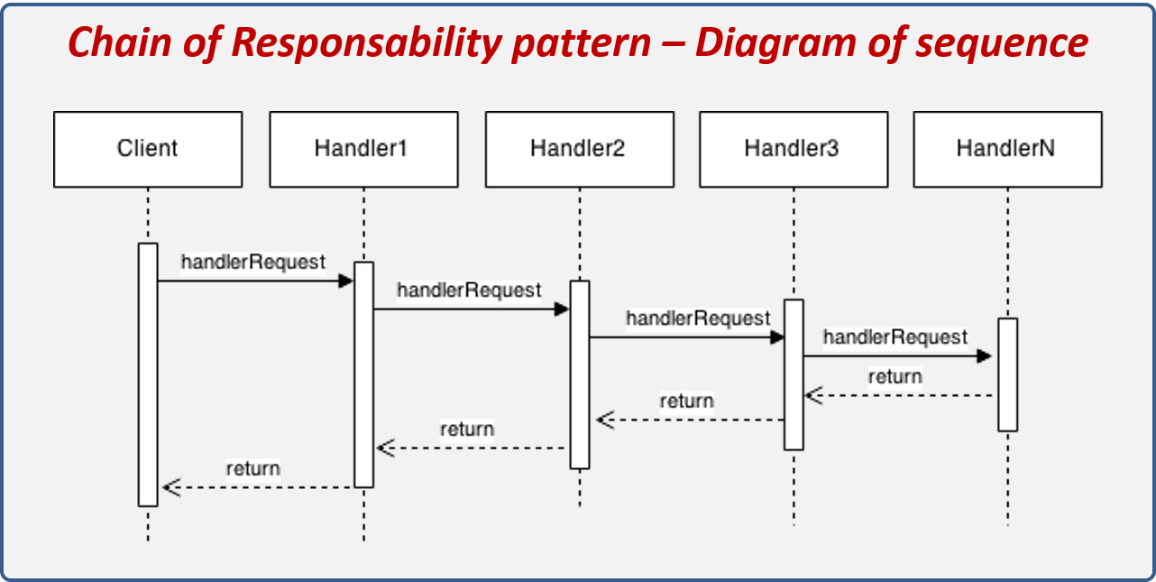
} else if …

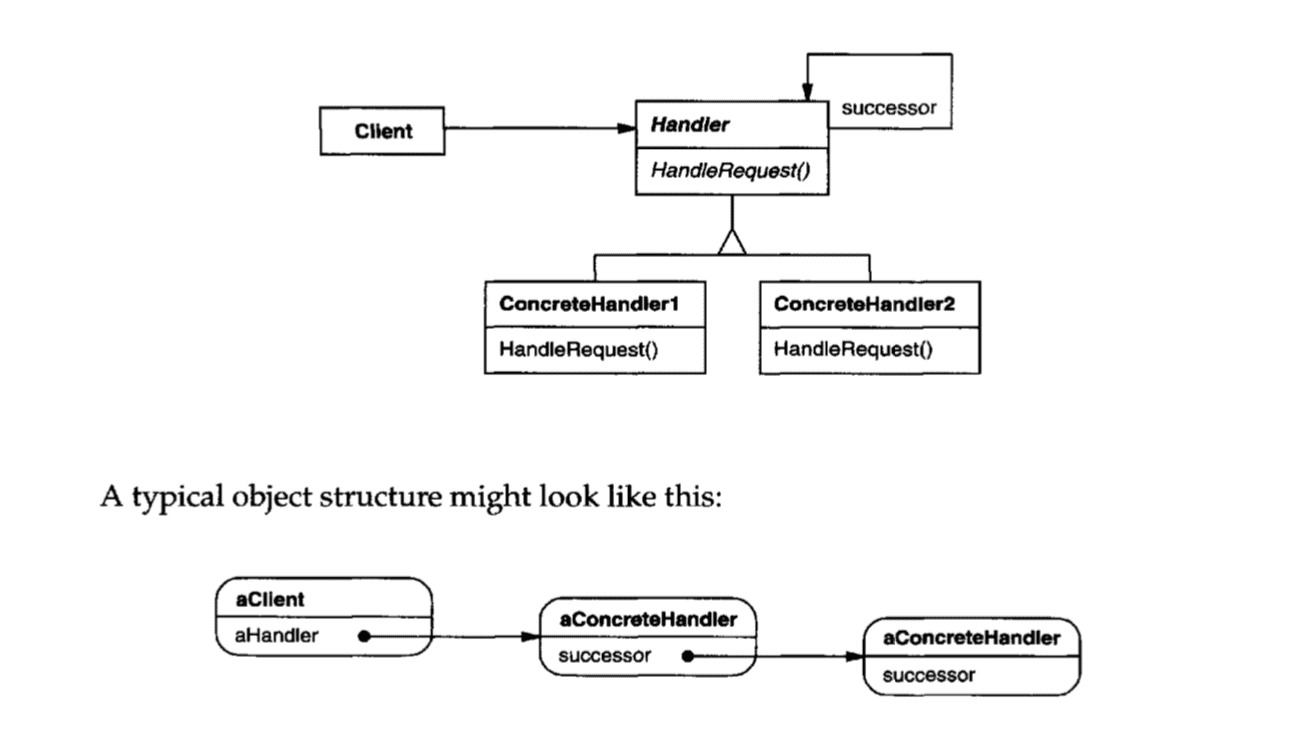
}

* **Chain of Responsibility 职责链 -对象行为型模式**

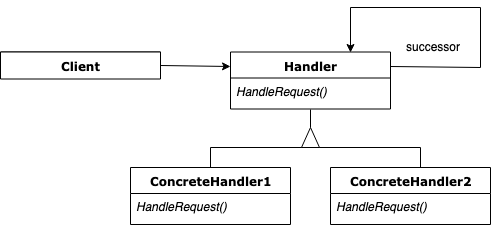
使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。只需要将这些对象连成一条链，并沿着这条链传递该请示。

要沿链转发请求，并保证接收者为隐式的，每个在链上的对象都有一致的处理请求和访问链上后继者的接口





ConcreteHandler处理它所负责的请求，可访问它的后继者，如果可处理该请求，就处理之，否则将该请求转发给它的后继者

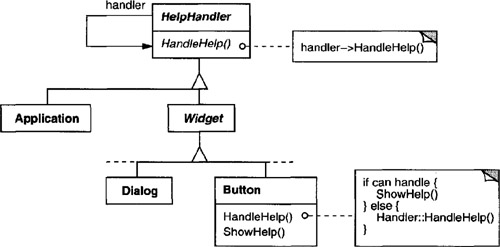


例：

考虑图形用户界面中的上下文帮助机制

应根据普遍性generality即从最特殊到最普遍的顺序来组织帮助信息。

Button -> Dialog -> Application



typedef int Topic;

const Topic NO\_HELP\_TOPIC = -1;

class HelpHandler {

public:

HelpHandler(HelpHandler\* = 0, Topic = NO\_HELP\_TOPIC);

virtual bool HasHelp();

virtual bool SetHandler(HelpHandler\*, Topic);

**virtual void HandleHelp();**

private:

**HelpHandler\* \_successor;**

Topic \_topic;

};

HelpHandler::HelpHandler(HelpHandler\* h, Topic t) : \_successor(h), \_topic(t) {}

bool HelpHandler::HasHelp() { return \_topic != NO\_HELP\_TOPIC; }

**void HelpHandler::HandleHelp() { if (\_successor != 0) \_successor->HandleHelp(); }**

class Widget : public HelpHandler {

protected:

Widget(Widget\* parent, Topic t = NO\_HELP\_TOPIC) : HelpHandler(parent, t) { \_parent = parent; }

private:

Widget\* \_parent;

};

class Button : public Widget {

public:

Button(Widget\* parent, Topic t = NO\_HELP\_TOPIC) : Widget(parent, t) { \_parent = parent; }

**virtual void HandleHelp();**

};

void Button::HandleHelp() {

If (HasHelp()) {

// offer help on the button

} else {

**HelpHandler::HandlerHelp();** // will call parent->HandleHelp()

}

}

class Dialog : public Widget {

public:

Dialog (HelpHandler\* h, Topic t = NO\_HELP\_TOPIC) : Widget(0) { SetHandler(h, t); }

virtual void HandleHelp();

};

void Dialog::HandleHelp() {

If (HasHelp()) {

// offer help on the dialog

} else {

HelpHandler::HandlerHelp();

}

}

class Application : public HelpHandler {

public:

Application (Topic t) : HelpHandler(0, t) { }

virtual void HandleHelp();

};

void Application::HandleHelp() {

// show a list of help topics

}

Button\* button = new Button(…);

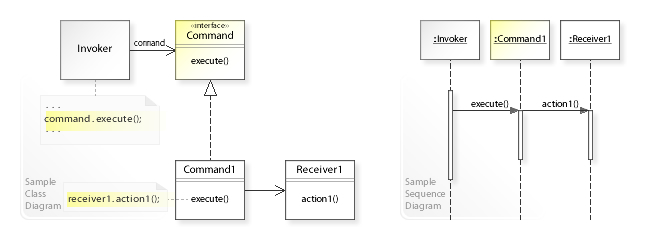
button->HandleHelp();

* **Command模式 （动作Action, 事务Transaction）**

**将请求封装在对象中**，这样它可作为参数来传递，也可以被存储在历史列表里，或者以其他方式使用。

**回调函数**是指函数先在某处注册，而它将在稍后某个需要的时候被调用；Command模式是回调机制的一个面向对象替代品。

该模式关键是Command抽象类，定义一个执行操作的抽象接口Execute()，具体的Command子类将接收者作其实例变量，并实现Execute(),指定接收者采取的动作



Command1将一个接收者对象receiver1绑定于一个动作，从而在execute()调用接收者操作

例：

VTK里的观察者/命令模式(Observer/Command)

监听感兴趣的事件，自动调用回调函数，执行相关操作

class vtkObject {

public:

int AddObserver(int eventId, vtkCommand\*);

};

客户端：

采用回调函数方式

void CallBackFun(vtkObject\*, int eventide, void\* data, void\* calldata) {

// …

}

vtkCallbackCommand mouseCallback = vtkCallbackCommand::New();

mouseCallback->SetCallback(CallBackFun) ; //设置事件回调函数

采用Command子类模式

class CallbackCommand : public vtkCommand {

public:

static CallbackCommand\* New() { return new CallbackCommand ; }

void SetObject(vtkConeSource\* cone) { m\_cone = cone; }

virtual void Execute(vtkObject\* caller, int eventide, void\* callData) {

// …

}

private:

vtkConeSource\* m\_cone;

};

CallbackCommand mouseCallback = CallbackCommand::New();

mouseCallback ->SetObject(cone);

// 注册鼠标左键按下事件，调用自定义

win->AddObserver(vtkCommand::LeftButtonPressEvent, mouseCallback) ;

# 架构模式

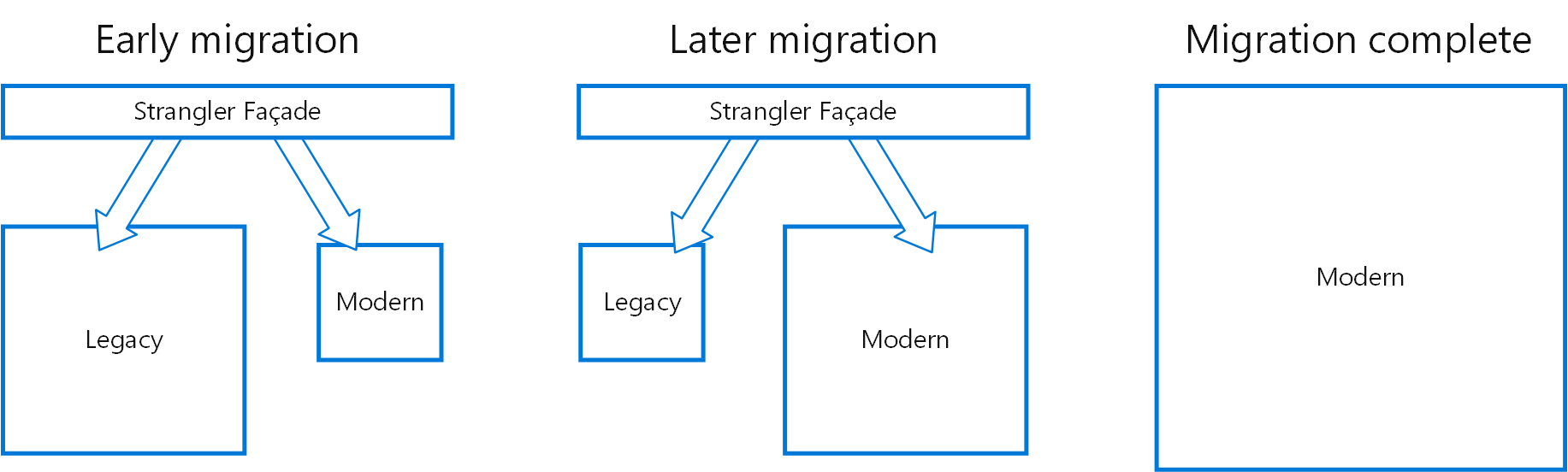
领域驱动设计，持续交付理论，六边形架构理论，按需虚拟化，基础设施自动化，小型自治团队，大型集群系统

## Back-end迁移到新架构 --Strangler Fig pattern 绞杀者模式

<https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig>

Context and problem: a gradual migration to a new system, while keeping the old system to handle features that haven't been migrated yet.

**Solution**: Incrementally replace specific pieces of functionality with new applications and services. Create a façade that intercepts requests going to the backend legacy system. The façade routes these requests either to the legacy application or the new services. Existing features can be migrated to the new system gradually, and consumers can continue using the same interface, unaware that any migration has taken place.



Issues and considerations:

1. Consider how to handle services and data stores that are potentially used by both new and legacy systems. Make sure both can access these resources side-by-side.
2. Structure new applications and services in a way that they can easily be intercepted and replaced in future strangler fig migrations.
3. At some point, when the migration is complete, the strangler fig façade will either go away or evolve into an adaptor for legacy clients.
4. Make sure the façade keeps up with the migration.
5. Make sure the façade doesn't become a single point of failure or a performance bottleneck.

不合适：

1. When requests to the back-end system cannot be intercepted.
2. For smaller systems where the complexity of wholesale replacement is low.

Layer分层

= view layer表现层 - business layer业务层 - persist layer数据访问层

表现层负责用户体验/交互，业务层负责具体的业务实现，数据访问层负责操作具体的数据库。

规范每层的接口。如果对第J+1层来说，第J层应该算是“黑盒”，就应设计一个统一接口，该接口提供第J层的所有服务，还可能将其封装到一个Facade对象中。

为支持自下而上的通信，可使用回调，并保留自上而下的单向耦合。使用回调时，上层向下层注册回调函数。在下层向上层发送的事件固定时，这种方式特别有效。

Pipeline/Filters

Omnilearn, VTK

Pipeline = Data stream -> step ->

MVC(Model-View-Control) & MVVM (Model-View-ViewModel)

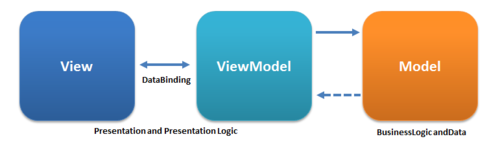
* MVC

M是指业务模型，V是指用户界面，C则是控制器，使用MVC的目的是将M和V的实现代码分离，从而使同一个程序可以使用不同的表现形式。

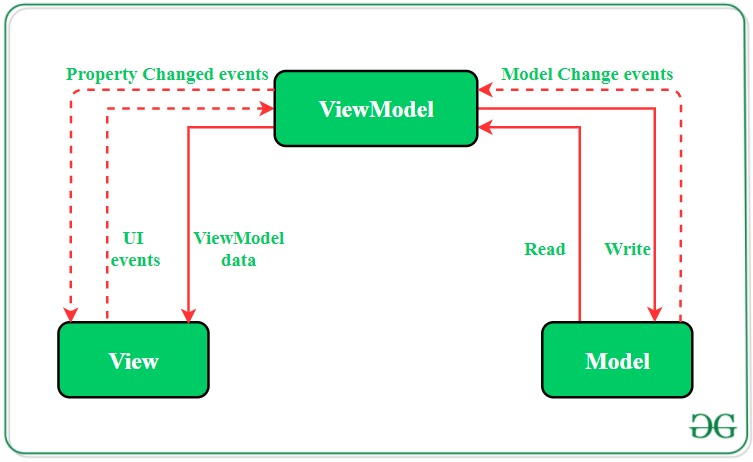
在MVC的三个部件中，模型拥有最多的处理任务, 在视图中其实没有真正的处理发生，它只是作为一种输出数据并允许用户操作的方式。C即controller控制器是指控制器接受用户的输入并调用模型和视图去完成用户的需求，它只是接收请求并决定调用哪个模型构件去处理请求，然后再确定用哪个视图来显示返回的数据。

* **MVVM: Model-View-ViewModel**

Common in Android



The viewmodel of MVVM is a value converter,[1] meaning it is responsible for exposing (converting) the data objects from the model in such a way they can be easily managed and presented. In this respect, the viewmodel is more model than view, and handles most (if not all) of the view's display logic.[1] The viewmodel may implement a mediator pattern, organizing access to the back-end logic around the set of use cases supported by the view.



Model: This layer is responsible for the abstraction of the data sources. **Model and ViewModel work together to get and save the data**.

View: The purpose of this layer is to inform the ViewModel about the user’s action. This layer observes the ViewModel and does not contain any kind of application logic.

ViewModel: It exposes those data streams which are relevant to the View. Moreover, it serves as a link between the Model and the View.

* **Monolithic（单体式开发）**

所有的功能打包在一个 WAR包里，基本没有外部依赖（除了容器），部署在一个JEE容器（Tomcat，JBoss，WebLogic）里，包含了 DO/DAO，Service，UI等所有逻辑。

客户端 - 服务提供API + DB

优点： 集中式管理，功能都在本地，没有分布式的管理和调用消耗

缺点：

1、效率低：开发都在同一个项目改代码，相互等待，冲突不断

2、维护难：代码功功能耦合在一起，新人不知道何从下手

3、不灵活：构建时间长，任何小修改都要重构整个项目，耗时

4、稳定性差：一个微小的问题，都可能导致整个应用挂掉

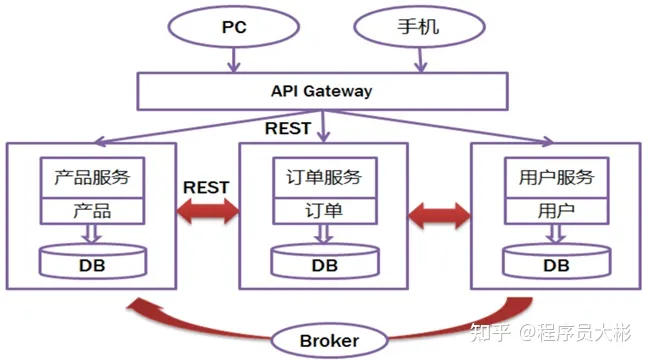
5、扩展性不够：无法满足高并发下的业务需求

* **微服务**

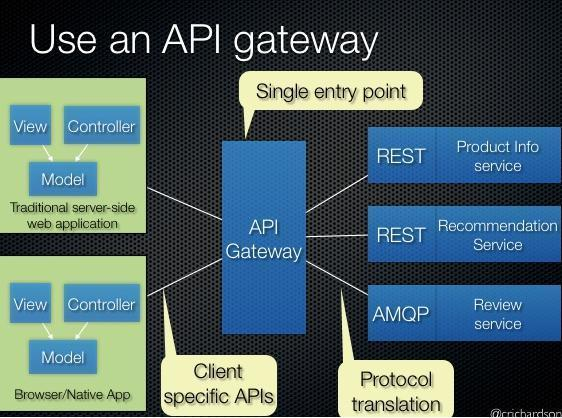
通过将应用和服务分解成更小的、松散耦合的组件，它们可以更加容易升级和扩展

特点：一个服务就是一个进程，运行在docker容器里，分布式管理，可以尝试最新的技术和语言和框架，然 后单独布署

服务粒度的粗细如何规划？



BAMA微服务有登录信息和权限管理服务，患者信息服务，营养模块服务等



客户端如web, mobile native app, …

客户端如何访问这些服务？通过API Gateway（提供统一服务入口，让微服务对前台透明）,

每个服务之间如何通信？微服务提供的RESTful API

如何管理服务？基本都是通过框架如zookeepe做**服务注册信息的分布式管理**。当服务上线时，服务提供者将自己的服务信息注册到框架，并通过心跳维持长链接，实时更新链接信息。服务调用者通过ZK寻址，根据可定制算法， 找到一个服务，还可以将服务信息缓存在本地以提高性能。

服务挂了，如何解决？

Monolithic方式开发一个很大的风险是，把所有鸡蛋放在一个篮子里，一荣俱荣，一损俱损。而分布式最大的特性就是网络是不可靠的。通过微服务拆分能降低这个风险，

当我们的系统是由一系列的服务调用链组成的时候，我们必须确保任一环节出问题都不至于影响整体链路。相应的手段有很多：①重试机制 ②限流 ③熔断机制 ④负载均衡⑤降级（本地缓存）

Front-end

Vue (基于 JavaScript的开源框架)，适合轻量级Web 应用程序或跨平台桌面应用程序，分Vue PC端框架和移动端UI框架。

Qt (基于C++的跨平台框架) ，用于构建桌面/移动应用程序和嵌入式系统。适合复杂的高性能的本地应用程序。

Back-end

## 微服务

Books: 《微服务设计》，《refactoring databases》

SOA：面向服务的架构，包含很多服务，而服务之间通过配合最终会提供一系列功能。**一个服务通常以独立的形式存在于操作系统进程中。服务之间通过网络调用**，而非采用进程内调用的方式进行通信

实施SOA时会遇到这些问题：通信协议（例如SOAP）如何选择，第三方中间件如何选择，服务粒度如何确定等。现有的SOA知识并不能帮助你把很大的应用程序划小。它没有提到多大算大，也没有讨论如何在现实世界中有效地防止服务之间的过度耦合。

微服务架构是SOA的一种特定方法：

1. **微服务是一种分布式系统**解决方案，推动细粒度服务的使用，这些服务协同工作，且每个服务都有自己的生命周期。
2. **微服务就是一些协同工作的小而自治的服务，根据业务的边界来确定服务的边**界。细粒度的微服务架构可以帮助他们更快地交付软件，并且有更多机会尝试新技术
3. 微服务就是一个独立的实体。它可以独立地部署在PaaS，也可以作为一个操作系统进程存在。服务之间均通过网络调用进行通信，从而加强了服务之间的隔离性，避免紧耦合。**服务应该可以彼此间独立进行修改**，并且某一个服务的部署不应该引起该服务消费方的变动。对于一个服务来说，我们需要考虑的什么应该暴露，什么应该隐藏。**如果暴露过多，那么服务消费方会与该服务的内部实现产生耦合**。这会使得服务和消费方之间产生额外的协调工作，从而降低服务的自治性。**解耦系统的黄金法则：你是否能够修改一个服务并对其进行部署，而不影响其他任何服**务
4. **服务通过API通信**：API的实现技术应该避免与消费方耦合，这就意味着应该选择与具体技术不相关的API实现方式，以保证技术的选择不被限制。

**微服务与共享库，模块等区别？**

共享库和模块缺点：限制采用新技术和独立对服务进行扩展的能力，并且有可能会导致使用过度耦合的集成技术，同时也会缺乏受益人接缝来进行架构的安全性保护；**共享库和模块会迅速和其他代码耦合在一起**，从而失去意义，而进程边界的存在则能够有效地避免这种情况的发生。

重用代码优缺点：重用代码，会引入服务之间的耦合。若把重用代码模板复制到各个服务中，如果核心服务代码模板升级了，那么需要花很长时间把这些升级应用到整个系统中。

**微服务优点：**

1. 技术异构性

在一个由多个服务相互协作的系统中，可以在不同的服务中使用最适合该服务的技术。比如性能提升部分，可以用C++，GPU…技术栈重新构建该部分。不同的数据存储技术比如对社交网络来说，图数据库能够更好地处理用户之间的交互操作，但是对于用户发布的帖子而言，文档数据库可能是一个更好的选择。

不过为了同时使用多种技术，也需要付出一些代价。所有有些组织会限制语言的选择，比如Netflix and Twitter选用的技术大多基于JVM

1. 弹性

在单块系统中，如果服务不可用，那么所有功能都会不可用。微服务系统本身就能够很好地处理服务不可用和功能降级问题。

1. 扩展

庞大的单块服务只能作为一个整体进行扩展。即使系统中只有一小部分存在性能问题，也需要对整个服务进行扩展。如果使用较小的多个服务，则可以只对需要扩展的服务进行扩展，这样就可以把那些不需要扩展的服务运行在更小的、性能稍差的硬件上。

1. 简化部署

在有几百万代码行的单块应用程序中，即使只修改了一行代码，也需要重新部署整个应用程序才能够发布该变更。于是在实际操作中，部署的频率就会变得很低；**在微服务架构中，各个服务的部署是独立的，这样就可以更快地对特定部分的代码进行部署**。如果真的出了问题，也只会影响一个服务，并且容易快速回滚。

1. 与组织结构相匹配

微服务架构可以很好地架构与组织结构相匹配，**避免出现过大的代码库，从而获得理想的团队大小及生产力。**

1. 可组合性

分布式系统和面向服务架构声称的主要好处是易于重用已有功能。现在的应用程序种类包括web, 原生应用，移动端web, 平板应用及可穿戴设备等。在微服务架构中，系统中会开放很多接缝供外部使用。当情况发生改变时，可以使用不同的方式构建应用，而整体化应用程序只能提供一个非常粗粒度的接缝供外部使用。

## 演化式架构师

架构师的一个重要职责是，确保团队有共同的技术愿景，以帮助我们向客户交付他们想要的系统。

架构师要对很多事情负责：确保指导开发的原则与组织的战略相符；确保原则指导的实践不会给开发人员带来痛苦； 了解新技术，需要知道在什么时候做怎样的取舍； 需要花时间和团队一起工作，甚至编码。但不应该独立做这些事情，可以由一个治理小组来做这个工作。一个小组通常会比单个人更加聪明。

对于我们创造的大多数产品来说，交付到客户手里之后，还是要响应客户的变更需求，而不是简单地交给客户一个一成不变的软件包。因此架构师必须改变那么从一开始就要设计出完美产品的想法，相反我们应该设计出一个合理的构架，在这个框架下可以慢慢演化出正常的系统。

**架构师类比城市规划师**，职责是优化城镇布局，使其更易于现有居民生活，同是也会考虑一些未来的因素。

作为架构师，不应该过多关注每个区域内发生的事情，而应该多关注区域之间的事情。

**每一个服务内部可以允许团队自己选择不同的技术栈或者数据存储技术**

代码架构师：架构师需要花时间和团队在一起工作，理想情况下他们应该一起进行编码，每周半天

架构原则：Heroku的12Factors帮助在Heroku平台上创建应用的设计原则

设计和交付实践：代码规范，日志数据集中捕获或http/rest作为标准集成风格等

监控有两种实现方式：推送机制（每个服务主动把数据推送到某个集中的位置）； 轮询系统（从各个节点收集数据）

**必须保证每个服务都可以应对下游服务的错误请求**。没有很好处理下游错误请求的服务越多，我们的系统就会越脆弱

范例，如果在系统中人们有比较好的代码范例可以模仿，那么他们也就不会错得很离谱

技术债务：短期可能会带来利益，长期要付出代价。不光走捷径会引入技术债务，有时候系统的目标会发生改变，并且与现有的实现不符，这种情况也会产生技术债务

建设团队：在单块系统中，人们为某些事情负责的机会非常有限，而在微服务架构中存在多个自治的代码库。

## 如何建模服务

好服务标准：**松耦合**（使用微服务最重要的一点是能够独立修改及部署单个服务而不需要修改系统的其他部分，一个松耦合的服务应该尽可能少地知道与之协作的那些服务的信息），**高内聚**（找到问题域的边界就可以确保相关的行为能放在同一个地方，并且它们会和其他边界以尽量松耦合的形式进行通信）

限界上下文：任何一个给定的领域包含多个限界上下文，每个限界上下文中的模型分成两部分：一部分不需要与外部通信，另一部分则需要。每个上下文都有明确的接口，该接口决定了它会暴露哪些模型给其他的上下文。

**应该共享特定的模型，而不应该共享内部表示**

对于一个新系统而言，可以先使用一段时间的单块系统，因为如果服务之间的边界搞错了，后面修复的代价会很大。所以最好能够等到系统稳定下来之后，再确定把哪些东西作为一个服务划分出去。

当你在思考组织内的限界上下文时，不应该从共享数据的角度来考虑，而应该从这些上下文能够提供的功能来考虑。**常常这么问：“这个上下文是做什么用的？”然后考虑“它需要什么样的数据”**

如果把系统分解成为限界上下文来表示领域的话，那么对于某个功能所要做的修改，就更倾向于局限在一个单独的微服务边界之内。这样就减小了修改的范围，并能够更快地进行部署。

## 集成

原则：**避免破坏性修改，保证API的技术无关性，使你的服务易于消费方使用，隐藏内部实现细节，为用户创建接口**

添加新客户可能会触发一个新的流程，比如进行付账设置，发送欢迎邮件等

应该使用数据库集成方式吗？

使用这种方式时，如果其他服务想要从一个服务获取信息，可以直接访问数据库，如果想要修改，也可以直接在数据库中修改。服务之间很容易通过数据库集成来共享数据，但是无法共享行为。内部表示暴露给了我们的消费方，而且很难做到无破坏性的修改。

服务之间的通信应该是同步还是异步？

如果使用同步通信，发起一个远程服务调用后，调用方会阻塞自己并等待整个操作的完成。如果使用异步通信，调用方不需要等待操作完成就可以返回，甚至可能不需要关心这个操作完成与否。

**异步通信好处如对于移动网络及设备而言，发送一个请求之后假设一切工作正常（除非被告知不正常），这种方式可以在很大程度上保证在网络很卡的情况下用户界面依然很流畅。**

基于请求/响应还是基于事件？（同步或异步通信都有这两种方式）

对于请求/响应，客户端发起一个请求，然后等待响应。这种模式与同步通信模式很好匹配，异步通信也可以使用这种模式，发起一个请求，然后注册一个回调，当服务端操作结束之后，会调用该回调

对于使用基于事件的协作方式，情况会颠倒过来。客户端不是发起请求，而是发布一个事件，然后期待其他的协作都接收到该消息，并且知道怎么做。**基于事件的系统天生就是异步的。业务逻辑并非集中存在于某个核心大脑，而是平均地分布在不同的协作者中。基于事件的协作方式耦合性很低。**客户端发布一个事件，但并不需要知道谁或者什么会对此做出响应，这也意味着，你可以不影响客户端的情况下对该事件添加新的订阅者。

编排与协同？

创建新客户的编排解决方案：客户服务作为中心大脑，在创建时它会跟积分帐户、电子邮件服务及邮政服务通过请求/响应的方式进行通信。

客户服务

积分账户

邮局服务

电子邮件服务

客户服务

积分账户

邮局服务

电子邮件服务

客户创建事件

发布

订阅

左图是通过编排处理客户创建，右图是通过协同处理客户创建事件

编排方式缺点：客户服务作为中心控制点承担了太多职责，它会成为网状结构的中心枢纽及很多逻辑的起点。我发现大多数重量级的编排方案都非常不稳定且修改代价很大。

通过来讲，我认为使用协同的方式可以降低系统的耦合度，并且你能更加灵活地对现有系统进行修改。但是确实需要额外的工作来对业务流程做跨服务的监控。

同步调用比较简单，而且很容易知道整个流程的工作是否正常。如果想要请求/响应风格的语义，又想避免其在耗时业务上的困境，可以采用异步请求加回调的方式。另一方面，使用异步方式有利于协同方案的实施，从而大大减少服务间的耦合。

**数据库集成->远程过程调用->REST？**

远程过程调用允许你进一个本地调用，但事实上结果是由某个远程服务器产生的。Java服务暴露SOAP接口，然后使用WSDL（web service definition language）定义的接口生成.NET客户端的代码。那么RPC的实现会帮你生成服务端和客户端的桩代码，从而让你快速开始编码。基本不用花时间，我就可以在服务之间进行内容交互了。

缺点：**技术的耦合**，RPC的性能问题（**RPC的核心想法是隐藏远程调用的复杂性，但是隐藏有点过头**，使用本地调用不会引起性能问题，但是RPC会花大量的时间对负荷进行封装和解封装，更别提网络通信所需要的时间）

相对于数据库集成，RPC显示是一个巨大的进步。

REST概念

资源，一个资源的对外显示方式和内部存储方式之间没有什么耦合

HTTP实现方式（HTTP也用来实现RPC，比如**SOAP就是基于HTTP进行路由，但只用到HTTP很少的特性，而动词和HTTP的错误码都被忽略**了

HATEOAS:超媒体作为程序状态的引擎：客户端应该与服务端通过那些其他资源的链接进行交互，而这些交互有可能造成状态转移。使用这些链接来对客户端和服务端进行解耦，从长期来看有着很显著的好处，因为你不需要一再调整客户端代码来匹配服务端的修改。通过使用这些链接，客户端能够自行获取相关API,这对于实现新客户端来说非常方便.我个人很喜欢让客户端自己遍历和发现API这种形式。自行发现和解耦的好处非常之大。

JSON, XML or HTML响应格式？

HTML既可以做UI，也可以做API,当然这么做是很容易出错的，因为与人类之间的交互，和与计算机之间的交互的差异是很大的。

接口约定？

避免将存储的数据直接暴露给消费者，在我的团队中一个很**有效的模式是先设计外部接口，等到外部接口稳定之后再实现微服务内部的数据持久化**。这样做可以保证服务的接口是由消费者的需求驱动出来，从而避免数据存储对外部接口的影响.

缺点：从易用性角度，RPC可以生成客户端的桩代码；对于低延迟的场景，每个HTTP请求的封装开销可能是个问题；

响应式扩展应用：简单地对下游服务调用的结果进行观察，而不需要关心它是阻塞的还是非阻塞的，唯一需要做的就是等待结果并做出响应

微服务代码重用的危险？

共享代码就有可能会导致这种耦合。比如客户端可以通过库共享其中表示系统核心实体的公共领域对象，而所有的服务也会使用这个库。所以当任何部分需要对库做修改时，都会引起其他部分的重新部署。

跨服务共同代码很有可能会引入耦合。但使用像日志库这样的公共代码就没什么问题，因为它们对外是不可见的。可以使用很多深度定制化的服务模板来快速创建新服务。他们不会在服务之间共同代码，而是把这些代码复制到每个新的服务中，以防止耦合的发生。

我的经验是：**在微服务内部不要违反DRY, 但在跨服务的情况下可以适当违反DRY**.服务之间引入大量的耦合会比重复代码带来更糟糕的问题。

如果开始服务端API和客户端API的是同一批人，那么服务端的逻辑就有可能泄露到客户端中。

容错性读取器：一些强类型语言会使用一些绑定技术，这种技术会将所有字段进行自动绑定，无论消费者是否需要。可以使用XPath来从中提取出想要的信息，这样字段的位置就可以更加灵活。这种读取器的实现能够忽略我们不在乎的那些修改。

**Postel法则（鲁棒性法则）**：客户端尽可能灵活地消费服务响应，系统中的每个模块都应该”宽进严出“，即**对自己发送的东西要严格，对接收的东西则要宽容**。该原则可以帮助我们**在服务发生改变时，减少消费方的修改**。

扩展/收缩模式：允许我们对破坏性修改进行平滑的过度。首先扩张服务的能力，对新老两种方式都进行支持。然后等到老的消费者都采用了新的方式，再通过收缩API去掉旧的功能。

用户界面

毕竟用户界面是连接各个微服务的工具，而只有把各个服务集成起来才能真正地为客户创造价值。分胖客户端和瘦客户端

很多公司会倾向把API设计得比较细粒度化，比如使用微服务架构所暴露出来的那些API。通过把服务的功能进行不同的组合，可以为桌面应用程序、移动端设备、可穿戴设备的客户提供不同的体验。

注意不同的客户端与系统之间，不同的交互会带来一些约束，比如移动端的约束

对与后端交互比较频繁的界面及需要给不同设备提供不同内容的界面来说，**一个常见的解决方案是，使用服务端的聚合接口或API入口。该入口可以对多个后端调用进行编排，并为不同的设备提供定制化的内容**。

与第三方软件集成：我和同事的建议是，对于一般规模的组织来说，如果某个软件非常特殊，并且它是你的战略性资产的话，那就自己构建，如果不是这么特别的话，那就购买

如何定制化？

CMS(内容管理系统)：工具设计上大多不支持持续集成，其提供的API非常难用，并且底层工具很小的升级都会破坏你做的那些定制化

Salesforce的问题尤其突出，这么多年来它一直在推Force.com平台，而这个平台需要使用一种叫做Apex的语言，该语言只能应用在Force.com的生态系统中

在自己可控的平台进行定制化

将CMS作为服务：从我的经验来看，CMS是一个最经常需要做定制化或者与集成的产品。因为除非你想要的是基本的静态站点，否则一般的企业都希望在自己的网站上提供动态内容，比如客户信息或最新的产品。CMS最常见的卖点是，你可以对其进行定制化，从而把各种特殊的内容放进来并显示给外部世界。然后普通的CMS开发环境通常都非常糟糕。普通的**CMS提供的主要功能是内容的创建与管理。大多数CMS甚至连页面布局都做不好**，它们通常只提供一些可拖拽的工具，然而这并不能满足你的需求，你还需要一些懂HTML和CSS的人来好好调整CMS模板。

Web前端

Web前端

Web前端

Web前端

第三方CMS

简化的API与内部格式进行匹配

更新内容

静态内容

CMS专用API调用

文档存储（外观）

使用CMS把你自己的服务隐藏起来

如何选择最大程度保证微服务之间的低耦合？

* 无论如何避免数据库集成
* 理解REST和RPC之间的取舍，但总是使用REST作为请求/响应模式的起点
* 相比编排，优先选择协同
* 避免破坏性修改，理解Postel法则，使用**容错性读取器**
* 将用户界面视为一个组合层

## 分解单块系统

《修改代码的艺术》，从接缝处可以抽取出相对独立的一部分代码，对这部分代码进行修改不会影响系统的其他部分。

首先创建包结构来表示这些上下文，然后把已有的代码移动到相应的位置，而哪些代码找不到合适的位置，这些剩下的代码很有可能就是我们遗漏掉的限界上下文。**Structure 101可视化包的依赖**。

分解数据库：**SchemaSpy图形化数据库之间的关系**。很多表最终会被分离到不同的限界上下文中，而上述的这个工具可以帮助你理解，这些横跨不同上下文的表之间存在什么样的耦合。那么如何切断这些连接呢？对于同一张表被多个限界上下文使用的场景又该如何处理呢？

**通过服务暴露API来访问数据，而不是直接访问数据库**，这个API调用会成为微服务化的第一步。我们可能需要实现跨服务的一致性检查，或者周期性触发清理数据的任务。这样做与否通常不由技术专家决定。

如何共享静态数据？如果所有的服务都要从同一张像国家这样的表中读取数据，该怎么办呢？

方法一：为每个包复制一份该表的内容，会导致潜在的一致性问题

方法二：共享的静态数据放入代码，比如放在属性文件中，或者简单地放在一个枚举中。数据一致性的问题仍然存在。但是**修改配置文件比修改在线数据库要简单得多**。通常这是比较合理的办法。

方法三：把静态数据放入一个单独的服务中。

从个人经验来看，**大部分场景下，都可以通过把这些数据放入配置文件或者代码中来解决问题**，而且它对于大部分场景来说都很容易实现。

如何共享数据？

领域概念不是在代码中进行建模，相反是在数据库中隐式地进行建模。这里缺失的领域概念是客户。需要把抽象的客户概念具象化。作为一个中间步骤，我们可以创建一个新的包，叫做Customer.然后让别的包如财务和仓库，通过API来访问此新创建的包。最终得到客户服务。

**重构数据库：推荐先分离数据库结构，暂时不对服务进行分离**，表结构分离之后，对于原先的某个动作而言，对数据库的访问次数可能会变多。先分离数据库结构但不分离服务的好处在于，可以随时选择回退这些修改或是继续做，而不影响服务的任何消费者。

如何保证事务？

最终一致性：相对于使用事务来保证系统处于一致的状态，最终一致性可以接受系统在未来的某个时间达到一致。**把未完成操作放在一个队列或日志文件中，之后再尝试对其进行触发，要保证重试能修复这个问题**。

补偿事务来抵消之前的操作。若补偿事务也失败，要么重试补偿事务，要么使用一些后台任务来清除这些不一致的状态。可以给后台的维护人员提供一个界面来进行该操作，或者将其自动化。

分布式事务：横跨多个事务，然后使用一个叫**事务管理器的工具**来统一管理其他底层系统中运行的事务。**处理分布式事务常用的算法是：两阶段提交**。首先是投票阶段，每个参与者会告诉事务管理器它是否应该继续。如果事务管理器收到的所有投票都成功，则会告知它们进行提交操作。只要收到一个否定的投票，事务管理器就会让所有的参与者回退。

**分布式事务很容易出错，而且不利于扩展**。通过重试和补偿达成最终一致性的方式，会使得定位问题更加困难，而且有可能需要其他的补偿措施来修复潜在数据的不一致

如果现在有一个业务操作发生在跨系统的单个事务中，那么问问自己是否真的需要这么做。是否可以简单地把它们放到不同的本地事务中，然后依赖于**最终一致性**的概念？这种系统的构建和扩展都会比较容易。

如果你遇到的场景确实需要保持一致性，那么尽量避免把它们放在不同的地方，一定要尽量这样做。如果实在不行，那么要**避免仅仅从纯技术（比如数据库事务）的角度考虑，而是显式地创建一个概念来表示这个事务**。

数据库报告

* 通过SQL查询对几张表做一个连接即可

通常为了防止对主系统性能产生影响，报告系统会从副本数据库中读取数据。标准的关系型数据库使用SQL作为查询接口，它能够和很多现成的报告工具协同工作。

* 通过服务调用来获取数据

缺点：为了从两个或者多个系统中获取数据，你需要进行多次调用，然后进行组装。对于某些服务暴露的资源来说，可以通过添加一些缓存头来加快数据的获取速度，但可能存在缓存无法命中。

* 极端方法：把对所有用户的请求建模成为一个资源，发起调用的系统可以POST一个BatchRequest,其中**携带一个位置信息，服务器可以将所有数据写入该文件**。

客户服务会返回HTTP202响应码表示请求已经接受了，但还没有处理。调用系统接下来轮询这个资源，直到得到一个201 Created状态，这表示请求已经被满足了，然后发起调用的系统就可以获取这个数据。通过这种方式可以将大数据文件导出，而不需要HTTP之上的开销，只是简单地把一个CSV文件存储到共享的位置而已。

除了报告系统拉取数据的方法，还可以把数据推送到报告系统中。使用标准的HTTP来进行大量调用时，会带来很大的额外开销，更不用提为报告系统创建专用API所带来的开销。一种替代方式是，使用一个独立的程序直接访问其他服务使用的那些数据库，把这些数据导出到单独的报告数据库（使用数据导出技术周期性地把数据推送到报告数据库）

事件数据导出：每个微服务可以在其管理的实体发生状态改变时发送一些事件。比如客户服务在客户增删改时发送一些事件。对于这些暴露事件聚合的微服务来说，可以编写自己的事件订阅器把数据导出到报告数据库中（基于状态改变事件来将事件数据导出到报告数据库中），**基于事件发生时就给报告系统发送数据，而不靠原有的周期性数据导出，所以数据就能更快地流入报告系统**。缺点：所有需要的信息都必须以事件的形式广播出云，所以在数据量比较大时，不容易像数据导出方式那样直接在数据库级别进行扩展。

## 部署

CI服务器会检测到代码已提交并签出，然后花些时间来验证代码是否通过编译以及测试能否通过。

是否真正在做CI的三个问题？

* 是否每天签入代码到主线
* 是否有一组测试验证修改
* 当构建失败后，团队是否把修复CI当作第一优先级的事情来做？

使用构建流水线建模的标准发布流程：

**编译并快速测试 -> 耗时测试 -> 用户验收测试 -> 性能测试 -> 生产环境**

在最开始的阶段，经常会发生跨服务边界的修改，所以时常会有些内容移入或者移出某个服务。在这个阶段，把所有服务都放在一个单独的构建中，可以减轻跨服务修改所带来的代码。

自动化可以对不同构建物的底层部署机制进行屏蔽。Chef, Puppet及Ansible都支持一些通过技术栈的构建物部署。

创建定制化虚拟机镜像：

启动普通的虚拟机 -> 应用版本管理控制的配置Puppet, Chef等 -> 从实例烧制镜像

不可变服务器：通过源代码管理的配置文件自动化重建环境和服务，但若部署完成，有人登录到机器上修改一些东西，这会导致机器上的实际配置和源代码管理的配置不再一致，这个问题叫配置漂移。为了避免这个问题，可以禁止对任何运行的服务器做手动修改。相反，**无论修改多么小，都需要经过构建流水线来创建新的机器。你甚至可以在镜像的创建过程中禁止SSH,以确保没有人能够登录到机器上做任何修改**。

服务与主机之间的映射？

单主机多服务：把多个服务绑定在一起进行部署，即部署全部的服务，这些工作都是为了简化单主机多服务的部署模型。但会放弃微服务的一个关键好处：独立部署不同的服务。

在单个主机上部署多个服务，会增加对单个服务进行扩展的复杂性。

每个主机一个服务：缺点是如何降低管理大量主机带来的额外负担？

**PaaS,能够帮你自己配置机器然后运行，能够透明地对系统进行伸缩管理，允许你控制运行服务的节点数量**，然后平台帮你处理其余的工作。不过平台一般都会尽量去满足一些比较通用的需求，而非特定用户的特殊需求，所以你的应用程序越不标准，就越难一起和PaaS进行工作。

传统的虚拟化技术：虚拟化技术允许我将一台物理机分成多台独立的主机，每台主机可以运行不同的东西。在OS上运行一个叫做hypervisor的东西，任务是两个，第一：对CPU和内存等资源做从虚拟主机到物理主机的映射。第二：给我们提供一个控制虚拟机的层

Vagrant：是一个很有用的部署平台，**通常在开发和测试环境时使用，而非生产环境。Vagrant可以在你的笔记本上创建一个虚拟的云**。它的底层使用的是标准的虚拟化系统(通常是VirtualBox)。缺点是开发机上会有很多额外的资源消耗。

Linux容器：可以创建一个隔离的进程空间。Linux内核的任务就是维护进程树。Linux容器扩展了这个想法，每个容器就是整个系统进程树的一棵子树。内核已经帮我们完成了给这些容器分配物理资源的任务。基本上所有的现代Linux内核都提供了LXC.

Docker是构建在轻量级容器之上的平台。它帮你处理了大多数与容器管理相关的事情。CoreOS是一个专门为Docker设计的操作系统。它是一个经过裁剪的LinuxOS, 仅提供有限的功能以保证Docker的运行。

**参数化的命令行调用是触发任何部署的最合理的方式**。可以使用CI工具来触发脚本的调用，或者手动键入。

环境定义：想像成微服务到计算、网络和存储资源之间的映射

测试

《敏捷软件测试》

《Scrum敏捷软件开发》

《测试驱动的面向对象软件开发》

单元测试：通常只测试一个函数和方法调用。单元测试是帮助我们开发人员的，是面向技术而非面向业务的。

服务测试：测试为用户界面提供服务的一些类。为了达到这种隔离性，我们需要给所有的外部合作者打桩，以便只把服务本身保留在测试范围内。

端到端测试：覆盖整个系统。这类测试通常需要打开一个浏览器来操作图形用户界面，也很容易模仿类似上传文件这样的用户交互。

测试比例，一个好的经验法则是，顺着金字塔向下，下面一层的测试数据要比上面一层多一个数量级。比如4000个单元测试，400个服务测试，40个端到端测试

测试场景而不是故事，把测试整个系统的重心放到少量核心的场景上来。把任何在这些核心场景之外的功能放在相互隔离的服务测试中覆盖

再怎么测试也不可能消除所有的缺陷，所以生产环境中有效的监控和恢复还是有必要的

跨功能测试

将系统拆分为较小的微服务后，跨网络边界调用的次数明显增加了。之前操作可能只涉及一次的数据库调用，现在可能涉及三四次跨网络边界来调用其他服务，还有匹配数量的数据库调用。由于性能测试运行的时间长，因此在每次构建的时候都运行性能测试并不是可行的。一个常见的做法是，每天运行一个子集，每周运行一个更大的集合。

监控

1. 监控主机。如CPU、内存等，可以使用New Relic这样的托管服务
2. 查看服务器本身的日志。
3. 监控应用程序本身。如服务的响应时间

如何在多个主机上的、成千上万行的日志中定位错误的原因？如何确定是一个服务器异常，还是一个系统性的问题？如何在多个主机间跟踪一个错误的调用链，找出引起这个错误的原因？

Logstash,解析多种日志文件格式，并将它们发送给下游系统进一步调查。

Kibana，基于elasticsearch查看日志的系统

Suro：Netflix的数据流水线，可以处理两种数据：用户行为的相关指标和更多的运营数据（应用程序日志）。然后这些数据可以被分发到不同的系统中，像Storm的实时分析、离线批处理的Hadoop或日志分析的Kibana.

安全

身份验证：确认他（主体）是谁，通常基于用户名和密码

授权机制：把主体映射到可以进行的操作。通常当一个主体通过身份验证后，我们将获得关于他的信息，这些信息可以帮助我们决定其可以进行的操作。

服务间的身份验证和授权：在边界内允许一切。边界内信任这种形式被大多数组织采用。

HTTPS基本身份验证，服务端需要管理自己的SSL证书

客户端证书：确认客户端采用TLS(Transport Layer Security安全传输层协议)，每个客户端都安装了一个X.509证书，用于客户端和服务器端之间建立通信链路

使用HTTPS路由通信缺点：除了需要管理证书，HTTPS通信的开销使得服务器压力增加，而且通信难以被轻松地缓存。

另一种方法：HMAC(Hash-based Message Authentication Code基于哈希的消息码)

使用HMAC,请求主体和私有密钥一起被哈希处理，生成的哈希值随请求一起发送。然后，服务器使用请求主体和自己的私钥副本重建哈希值。如何匹配，它便接受请求。这样做的好处是，如果一个中间人更改了请求，那么哈希值会不匹配，服务器便知道该请求已被篡改过。并且，私钥永远不会随请求发送。

HMAC缺点：客户端和服务器需要一个共享的、以某种方式交流的密钥。如何共享？两端硬编码？这是一种模式，而不是标准，因此有各种不同的实现方式。如JWT(json web tokens，JSON Web令牌)

静态数据的安全： 对静态数据的加密 ，使用众所周知的加密算法（因为经过同行评审多年，千万不要实现自己的加密算法，不要发明自己的安全协议）。加密的过程依赖加密算法和密钥。如何存储密钥？使用单独的安全设备来加密 和解密数据。另一个方案是，使用单独的密钥库，当你的服务需要密钥的时候可以访问它。

操作系统：用户尽量少的权限，更新最新的软件，防火墙，网络隔离

## 云服务模式

Books: 《让云落地—云计算服务模式》

**云模式：SaaS, PaaS and IaaS, 公有云私有云和混合云**

云计算资源采取了一种如同水电的按需付费的定价模式，我们可以很方便地在一个原型环境中以极小的投入来完成多个配置的测试工作

**商用云的设计初衷是将基础设施商品化**，并以较低的成本对外提供，使用户能够获得高扩展性和自服务能力。

成功例子：

Instagram照片分享应用发布， 无须管理数据中和网络，也无须采购、安装或管理硬件，只要专注于应用架构和用户体验这两项他们所擅长的事情即可。

Netflix互联网流媒体视频内容行业的领导品牌，专注于构建和提高业务应用（Netflix的核心竞争力）上，然后让亚马逊来负责基础设施（亚马逊的核心竞争力）的问题

云计算的五种特征：网络接入、弹性、资源池化、可计量的服务以及按需自服务

托管并不提供云计算的五个特征，托管只是在托管服务提供商处租用或购买基础设施和地面空间。

**公有云**：多租户环境，即最终用户与其他消费者一起，在一个共享的商业资源网络上为自己所使用的资源付费。物理硬件之上是一个抽象层，以API的形式展现给最终用户。优点：用户只为所消费的资源付费； 弹性（动态提升或降低其在处理峰值负载时所需的计算资源数量）。缺点：用户依赖公有云；数据隐私问题；

**私有云**优点解决公有云的缺点，在单一租房环境下进进部署，不会与其他用户混用。有本地私有云和托管的私有云。私有云牺牲了“快速伸缩性、资源池化以及按需使用的定价模式”这些云计算的核心优势.私有云可访问的资源总量取决于内部所购买和管理的基础设施的多少。

**混合云：**在利用快速伸缩性和资源池这些云计算的优势方面尽可能多地使用公有云，而在数据所有权和隐私这些公有云中风险较高的领域使用私有云

大企业会选择混合云服务，把数据保存在私有云里，然后把不重要的组件迁移到公有云中

**SaaS（软件即服务）**

SaaS是一种以服务形式向消费者交付的完整应用。服务消费者要做的只是对一些具体的应用参数进行配置和对用户进行管理。

**PaaS（平台即服务）**

将大部分标准化的应用堆栈层的功能抽象出来，将之以服务的形式对外提供。**消费者能够使用提供商所支持的编程语言、库、服务和工具，将自己创建或获取的应用部署到云基础设施上**。消费者不会对底层云基础设施进行管理或控制，这包括网络、服务器、操作系统或存储等，但是可控制包部署的应用，并有可能控制配置应用的托管环境。

PaaS服务提供商管理应用平台，向开发者提供一套工具来加快开发流程； 而开发者在使用PaaS服务时，由于受到这些工具和软件包的约束，在某种程序上要放弃一些灵活性。另外，**在一些较底层软件的控制上，比如内存分配或堆栈配置方面（例如：线程数、缓存容量、补丁级别等），开发者也几乎没有控制权限**

PaaS的另一个巨大优势就是，这些平台可以与大量第三方软件解决方案进行整合，也就是我们常说的插件plugins，附加组件add-ons或扩展extensions. 如数据库，日志，监控，安全，缓存，搜索，电子邮件，分析，支付。

**IaaS（基础设施即服务）**

**IaaS可以控制操作系统、存储、所部署的应用**，或者对网络组件如防火墙的选择有部分控制权，在IaaS中，涉及管理和维护物理数据中心和物理基础设施（服务器、磁盘存储、网络等）的许多工作，都被抽象成一系列可用服务。OpenStack是一个开源项目，构建私有云提供IaaS功能

**云堆栈：**

**基础设施层（IaaS解决的问题）**：硬件和周边设备、网络基础设施、存储设备等各种物理实体构成。

**应用堆栈层(PaaS解决的问题)**：保证应用软件安全的责任，比如操作系统、应用服务器、数据库软件和编程语言。提供应用堆栈工具如缓存、队列、消息、电子邮件、日志、监控之类的按需服务。

在消费者眼中，PaaS提供商负责所有的基础设施和应用堆栈安全。不过实际上，PaaS提供商管理应用堆栈的安全，但是会借助IaaS提供商来提供基础设施的安全。

**应用层：**如安全传输协议https/sFTP、加密数据、对用户认证和授权、防止网页漏洞之类的事项。

**选择合适的云服务模式？**

每种云服务模式都通过某种程度上的资源抽象，来减低消息费构建和部署系统的复杂性。正如建造房子需要许多不同的技能和许多不同的工具，一种云服务模式并不一定能解决所有问题

PaaS最大的优点之一：平台使底层基础架构对开发者透明，开发者从而可以专注于业务需求的实现，而**平台负责资源的自动伸缩**。由于PaaS供应商要负责满足所有租户的扩展需求，所以他们会将单一租户能够请求的资源数量限定在一定范围内。因此某些访问量在世界上数一数二的网站，比如Facebook,Twitter都会使用IaaS云服务模式

搭建社交媒体网站的公司在客户自愿发布其照片、视频等个人数据的情况下，可能会对实现大规模支撑和高运行时间的技术需求更为看重；负责处理医疗索赔的医疗公司很可能把风险这一类问题看得比什么都重要

非核心竞争力的方面都是使用SaaS方案的良好选择

要求开发者管理内存、配置数据库服务器和应用服务器，以最大化吞吐量、明确数据如何在磁盘之间分布和控制操作系统等，你就应该选择IaaS

**应用迁移至云端的注意事项：**

**云计算架构要求是的一种“松耦合”**，弹性意味着软件能够按需进行扩展或缩减，而且必须不受运行所处物理环境的限制。如果软件开发是在数年以前完成的，那么软件对其运行所处的物理硬件甚至开发使用的技术堆栈非常可能有着很高的依赖性。

**云服务是无状态的。**一个“无状态的” 服务是指服务不知道前一个请求或响应的任何信息，只知道服务处理给定请求这一持续期间的信息。无状态的服务在客户端 而非服务器端存储应用的状态，因此对基础设施没有依赖性。遗留应用程序常常是有状态

**云安全**取决于软件架构师如何加密数据、管理密钥以及实现良好的密码策略。如果有着适当的安全架构，公有云会比绝大多数本地数据中心更安全。

每种云服务都会带来某种程度上的供应商锁定。研究锁定的成因和影响，确定服务等级。拒绝从主机系统迁移至主机客户端模式，反对企业使用互联网，这种情形常常出现是因为当人们获知要进行改变时，通常他们的第一反应是拒绝改变。

云本身在安全性上并不比企业自己的数据中心差

云服务提供商环境下的攻击倾向于机会性犯罪，而企业数据中心里的攻击则目标更明确，也更有经验

在云和企业数据中心里，Web应用采到威胁的机会均等

IaaS提供商有义务保护所有租户的权利，所以不会允许某一个租户的审计人员进入其基础设施，公司的审计人员将只能对IaaS供应者制作的白皮书和发行的审计报告进行检查，并且无权访问公有的IaaS数据中心

数据的所有者是谁？开发软件的公司，提供数据的第三方，还是使用系统的客户？

**许多传统架构中的应用会依赖于“状态”，而构建云端软件需要开发无状态应用。如何转变？**

1. 云要求开发技能在系统管理方面表现出更高的优先级，许多手动管理任务现在可以以云服务的方式获得
2. 开发人员需要对网络、安全、分布式计算、SOA、网页架构等更多方面有着丰富的知识
3. 可以采用引入具有云经验的新员工来帮助公司转型并同时培养、提高现有的员工不失为一个好的策略。
4. 业务需求的细节应当决定采用何种**云部署模式（公有，私有，混合）**和何种**服务模式（IaaS, PaaS, SaaS）**来实现解决方案的架构。如果公司在建设面向消费者的网站，用户自愿交换个人数据以获得免费服务，则公有云；如果公司向类似医院或政府部门提供服务，要考虑安全，隐私等等，考虑混合云。
5. 初创企业需要在云中构建自己的全新应用即可；一个运营已久的企业将不得不对企业现有的每一个组件分别进行评估，以判断能将哪些迁移至云中，每个组件更适合哪部部署模式（公有，私有，混合）
6. 遗留系统迁移至云端的挑战：许多遗留系统都对**ACID事务(原子性，一致性，隔离性和持久性)**有依赖，在本地环境中，数据可能只绑定在一个分区之上；云架构依靠的是**BASE事务（基本上可用，软状态，最终一致性）**，BASE事务认可资源可能失效，但数据最终将会变得一致的方式

**云安全策略**

集中化：构建一组常见的服务来认证和授权用户使用云服务。云系统应该只提供有限的访问路径供人员或系统接入，客户应该全部通过同一个能被监控到的入口进入。

标准化：公司应该考虑对接入系统实施类似OAuth or OpenID的行业标准

自动化：通过对虚拟机和代码的部署实现脚本化，可以在需求增加或减少时自动进行资源的缩放

任何包含敏感数据并且通过互联网处理的消息都应采用诸如https、sFTP或SSL等安全协议

在数据库中，可以在属性级、行级或表级加密敏感数据。

存放数据的文件或目录可以进行加密。对文件的访问也可以设置密码保护，要求用密钥进行解密。

公私密钥对：公钥保护的任何对象，都只能通过相对应的私钥进行解密

不以明文存储密钥

Web安全：在传输过程中拦截数据、注入SQL语句、劫持用户会话以及完成各种恶意行为。

API管理：

1. 合作伙伴和客户共享API的公司来说，最好能使你的API支持OAuth和OpenID；
2. 客户和提供商之间尽量避免使用密码，而应使用API密钥；
3. 避免会话和会话状态来避免会话劫持的发生，在处理每一个请求时重置认证，这样即便一个未授权用户通过某种方式得到了认证，在请求结束之后他们就再也不能访问系统。

补丁管理：在使用IaaS时，云服务消费者对应用堆栈负责，因此必须对操作系统、数据库服务器、应用服务器、开发语言以及其他所有组成的系统软件和服务器的安全进行管理。对系统打补丁的行为应至少每30天完成一次。不建议将安全补丁应用到现有服务器上； 而应遵循创建新的、销毁旧的服务器的原则

监控指通过一套工具监视系统的过程，该套工具提供了有关系统健康及活动发生的信息。

**RESTful服务**

云是一种把以许多不同技术堆栈编写的、来自许多不同公司的、许多不同服务连接起来的异构的生态系统。

在云中编写松耦合软件的关键办法之一就是将应用的状态存放在客户端而非服务器端，这样就打破了硬件和软件之间的依赖性。

REST的四个约束：

1. 区分资源和表述：资源可能是数据存储或代码块，而表述可能是XML,JSON or HTML
2. 通过表述来操作资源
3. 自描述消息：每条消息都提供了足够多的信息来描述如何处理消息
4. 以超媒体作为应用状态的引擎

**数据集**

性能分为三类：实时（亚秒级响应）、近实时（1~2秒）以及延时。

常见的用于大容量高性能数据集的设计模式是：

1. 使用缓存层
2. 减少数据集的大小（存储属性的hash）
3. 将数据库区分为只读节点和只写节点
4. 将数据进行切片，分为特定客户切片、特定时间切片和特定区域切片
5. 对陈旧数据进行归档，降低表的大小
6. 对数据集进行非规范化处理

数据集分为静态数据集和动态数据集

静态数据集：通常是事件驱动的数据，按时间顺序发生。典型的例子是**Web日志、事务和收集数据**。此类静态数据集属于 “一次写入，多次读取 “类型的数据集，在某一个时间点发生，但是会反复读取分析以检测模式和观察行为，**占用TB级的数据空间，通常采用非标准化的数据库**操作来使性能最大化。挖掘这类数据集的常见操作是对数据进行非规范化处理、使用NoSQL数据库，以及应用大数据技术

动态数据集：**数据经常变化，规范化的关系型数据库管理系统是最常见的解决方案**。关系型数据库非常适合处理ACID原子性、一致性、隔离性和持久性事务以确保数据可靠性。

使用关系型数据库的好处很多，但**当数据容量达到某一规模时，关系型数据库会变得非常慢**。因为关系型数据的构建参照完整性。为了达到这一点，在数据库引擎中内置了大量的开销来确保在数据存放进表之间完成事务的处理和提交。

对数据库和文件系统进行备份是保证业务的可持续性和灾难恢复的关键。全备份通常会每天进行，而增量备份则会在一天的多个时间内进行。一种常见的策略是在从数据库上进行备份，这样应用程序的性能就不会受到影响

多租用意味着有多个组织或客户（租户）共享一组服务器；单租户意味着每组服务器只支撑一个租户。

**键值存储数据库：**采用hash表，每个带有指针的唯一的键都会指向一个特定的数据项。在读取类似历史订单、时间和交易这种大型、静态、结构化数据时其也有良好的表现。因为没有对数据库的结构描述，所以**不适合处理复杂数据和关系**

**列存储数据库：**为了存储和处理跨多台机器之上的大规模分布式数据

**文档存储数据库：**存储以文档形式进行存放的非结构化数据。数据通常以XML，JSON, PDF, Word, Excel和其他常见的文档类型进行封装

**图像数据库：**用于存储和管理彼此关联的关系。

**文件存储数据：**比如像照片、视频和MP3这样的大型文件。Web服务器如果用数据库对这些大字段进行存储和检索的话，可能很难提供高性能的用户体验。更好的方法是利用内容分发网络CDN这种通过互联网连接的位于多个数据中心的分布式计算机网络。CDN能提供高可能性和高性能，是流媒体和其他带宽密集型数据的选择工具之一。

不管数据是不是在云中，政府都能向任何公司提出数据请求。加密数据是降低政府数据请求风险、驳斥重要数据不能存放在公有云中的最好方法。

消费者越往云堆栈上层走，他们移交给提供商的责任也就越多。

**主要数据中心的灾难恢复：**

1. 经典的备份和恢复方法：每天完整备份和增量备份，存放到云供应商的磁盘服务
2. 冗余的数据中心—单活冷备：配备一组与运行在主数据中心上的资源配置完全一样的服务器，“冷“意味着冗余的服务器并没有启动和运行在主数据中心上的资源配置完全一样的服务器
3. 冗余的数据中心—单活温备：数据库服务器运行与主数据中心保持同步，但其他服务器处于冷状态
4. 冗余的数据中心—双活热备：所有服务器运行，完全冗余的数据中心

**创建集中化的日志策略**

日志：指的是所有系统记录的采集，对系统内发生的每个事件，尤其是有用户或系统请求访问的事件进行记录是一种很好的做法。

日志文件在数据库操作行为、用户访问、错误和调试信息等方面，很有用。同时管理员就可以移除开发者对所有生产服务器的访问权限。不允许开发人员和运营人员访问生产服务器。

一旦数据达到日志服务器，就有多个日志记录方案将数据转换成NoSQL数据库。这些工具还提供了丰富的用户界面功能，使最终用户可以搜索日志和计划作业、触发警报、创建报告等。

也可采用SaaS日志解决方案

标准化日志格式：将所有的日志格式、命名规范、严重级别以及所有消息的错误代码都进行标准化处理。标准对于优化搜索和输出一致性结果的意义非常明显。日志消息中包含的数据越标准，在设计时能进行的自动化程度越高。

使日志更容易访问、一致性、有意义、可搜索以及集中管理，是所有云方案的核心策略

**监控**

前端开发人员可能关心页面加载时间、网络性能、应用程序接口性能等

数据库架构师可能希望在SQL语句及其响应时间的指标之外，还看到线程、缓存、内存和CPU利用率等有关数据库服务器的各项指标。

系统管理员可能相要看到诸如每秒请求数、磁盘空间容量，以及CPU和内存利用率等指标

产品所有者或许想看到每天的不同访问者数量、新用户数、单用户成本以及其他业务相关指标

私有的PaaS更像是IaaS.对于两者，消费者都必须对低至应用堆栈层的系统进行监控

监控内容：性能和吞吐率

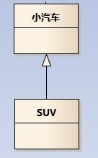
**用户性能指标**：新用户数，每天不同访问者的数量，每天的页面访问数量，平均站点停留时间，每客户收入，跳出率（未访问页面直接离开的用户占比），转换率（在直销的基础上完成预期操作的用户占比）

**响应时间性能**：页面加载时间，运行时间，响应时间(API、报告、查询等)

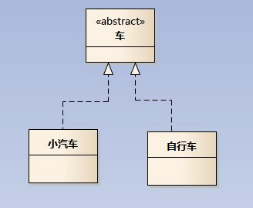
**吞吐率**：衡量了数据通过系统的平均速率。在用户层，吞吐率测量系统正在处理的并发用户数或会话数。在应用层，吞吐率测量的是系统能通过应用层从应用堆栈层向最终用户传送的数据量的大小 .

UML

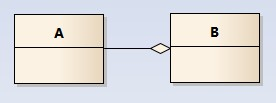
泛化关系generalization：表示is-a， 代码一般是继承非抽象类



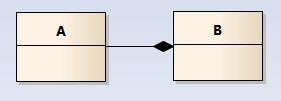
实现关系realize：表示is-a, 代码一般是继承抽象类



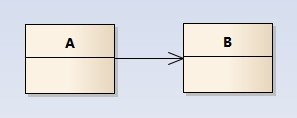
聚合关系aggregation：表示整体由部分构成（如一个部门由多个员工组成），与组合关系不同的是，整体和部分不是强依赖的，即使整体不存在了，部分仍存在（如部门撤销了，人员不会消失，他们依然存在）



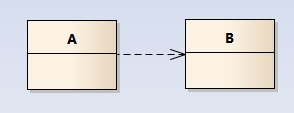
组合关系composition：强依赖的聚合关系，如果整体不存在了，则部分也不存在（如公司不存在，部门也不存在）



关联关系association：静态关系，与运行状态无关，一般由常识决定（如乘车人和车票是关联关系，学生与学校是关联关系），代码中以**成员变量**实现



依赖关系dependency：临时性关系 ，运行期间产生。代码中体现为类构造方法或类**方法的传入参数**。



# DevOps&Tools

一种文化转变，或者说是一种新的思考我们如何开发和发布软件的方式

目标是以开发、运维、质量保证的需求彼此相关，并且这些需求应该是一个协作过程的组成部分，这样一种心态来构建系统的

DevOps思维：理解工作流程，始终寻求提高流程的方法，不向下游传递缺陷，取得对系统的时刻理解。

**团队关注点：自动化基础设施，自动化部署**

持续集成CI：每次提交时构建和测试应用的行为

持续交付CD：除了CI，还在这个流程中加入了自动化测试和自动化部署

常见的初创企业：并没有发明新的事物，其选定一个当今遗留软件提供很差服务或根本没有提供服务的商业流程，然后使用新技术以一种服务的形式提供这种商业流程

分布式系统的延迟和容错库Netflix/Hystrix, Dropwizard（开发Java RESTful Web服务，同Flask）

HTTP缓存代理、mod\_proxy负载均衡器，大量针对HTTP的监控工具

RabbitMQ, ATOM, 响应式扩展(Reactive extensions, Rx)

Backends for frontends（为前端服务的后端）：允许团队在专注于给定UI的同时，也会处理与之相关的服务端组件。后端虽然嵌入在服务端，但它也是用户界面的组成部分

Structure 101，可视化包之间的依赖

SchemaSpy, 图形化数据表之间的关系

Packer, 借助工具Chef, Ansible, Puppet或者其他从同一套配置中生成不同平台的镜像

Jetty and Tomcat?

Deis, 开源的PaaS系统，基于Docker之上，提供类似Heroku那样的PaaS.

CoreOS，专门为Docker设计的操作系统，经过裁剪的LinuxOS, 仅提供了有限的功能以保证Docker的运行。

Python Fabric脚本，管理大量host比如运维，私有云管理，应用自动化部署等

Terraform, 自动配置负载均衡器…

CoreOS（基于Docker的LinuxOS）, Deis(基于Docker提供类似Heroku的PaaS)

Python Fabric，将命令行调用映射到函数的Python库，也提供类似SSH这样的机制控制远程机器

mBaaS: 移动后端即服务，只需要创建一个用户界面，mBaaS方案会负责将其设计进行转换适合市面上的多种设备使用。

Pact: 消费者驱动的测试工具。

# <企业集成模式：设计、构建及部署消息传递解决方案>

1. **文件传输**

一个应用写文件，之后另一个应用读这个文件。为此，应用之间需要协商文件名、文件的位置、文件的格式、文件读写的时间以及谁负责删除这个文件

1. **共享数据库**

多个应用共享相同的数据库，这个数据库位于独立的物理数据库中。

1. **远程过程调用**

一个应用开放其部分功能，使得其他应用能够远程访问这些过程。它们之间的通信是实时、同步的。

1. **消息传递**

一个应用向公共消息通道中发布一个消息，其他应用可以在之后某个时间从通道中获得这个消息。应用之间必须协商建立通道以及消息的格式

# <企业应用架构模式>

# Heroku

1. Your Heroku Account
2. Heroku Client

Install Heroku toolbelt <https://toolbelt.heroku.com/windows>

>heroku login

[Qizhong.lin@philips.com](mailto:Qizhong.lin@philips.com)

Password: same to google email

Deploy your application to Keroku

App>yo jhipster:heroku

mLab for mangodb

**uri:** mongodb://heroku\_55ckb2n1:qzlin19811106@ds023912.mlab.com:23912  
**database:** heroku\_55ckb2n1

# Pivotal Web Services

1. your PWS Account -------Apps Manager
2. setup PWS Account
3. create ORG and space
4. CF Client -------deploy and manage applications
5. Install CF Client
6. Connect Client to PWS

// private

>cf login –a <https://api.run.pivotal.io> ------Cloud Foundry public API endpoint UserID：[qizhong.lin@philips.com](mailto:qizhong.lin@philips.com)/qizhonglin157@sina.com

Password: same.to.google.email

Note: if connection fails, set HTTP\_PROXY=161.92.51.225:8080 in System Environment

1. Deploy an application to PWS

App> cf push

Manifest.yml

.war

1. Use application

Open route of application, it will return you web page(html+css+javascript), json or xml

Bind app to backing services (from Marketplace)?

（初步理解为：调用pivotal平台上已有的服务，如常见的有relational databases, MongoDB, Redis, RabbitMQ (message queue)）

技术点: Sprint Cloud



Scale app (supplied by platform)

1. Logs, Events, performance

How to monitor App Performance? (database call response time & throughout, performance data API access)

Bind app to Service (New Relic)

# Monolithic vs micro-service

monolithic application

(e.g. application = one huge WAR file)

micro-services

application as a collection of independently deployable services.

I have often introduced the idea as “SOA light” since you are building a service-oriented architecture. The trouble with the term SOA, however, is that it is associated with a lot of baggage: SOAP, ESBs, heavyweight ceremony, etc. Instead, I’ve talked about “Decomposing the monolith” or “Decomposing the WAR”. I’ve also used the term modular, polyglot architecture but that’s a bit of a mouthful.

Users expect a rich, interactive and dynamic user experience on a wide variety of clients including mobile devices. Applications must be highly scalable, highly available and run on cloud environments.

## Eureka = Netflix Service Discovery Server + Client

Eureka服务器没有存储功能，所有注册表里的每个服务（通过自己的application.yml的serviceUrl）每30秒发送一次心跳，表明自己活着。

Eureka客户端会缓存服务器注册表

// @EnableEurekaServer, @EnableDiscoveryClient运行Eureka服务器

Eureka服务器自带UI and HTTP API端口: localhost:8761 or localhost:8761/eureka/

默认情况下，每个Eureka服务器也是Eureka客户端，需要service URL访问peer

application.yml

server:

port: 8761

eureka:

client:

serviceUrl:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

instance:

hostname: localhost

//as standalone eureka server, 关闭客户端行为（因为服务器也是客户端）

client:

registerWithEureka: false

fetechRegistry: false

// @EnableEurekaClient （如下两个功能）

自动注册该服务于Eureka服务器中

application.yml

// eureka的基本信息会记录于服务器注册表中，从而别的服务可以访问到

eureka:

client:

serviceUrl:

defaultZone: <http://localhost:8761/eureka/> 定位Eureka服务器

instance:

hostname: localhost

nonSecurePort: 80

metadataMap:

instanceId: ${spring.application.name}:${spring.application.instance\_id:${random.value}}

通过Eureka服务器查询别的服务

## Communicate between front-end and back-end?

Emulator <-> web service directly

Device <-> web service, should in the same domain, fire-wall should in-bound port

1. Device browser access to web service

ip:port\ccs\patients\patient1

if can’t access, please do as follows:

control panel -> system and security -> windows firewall -> advance settings -> Inbound Rules

New Rules -> port -> TCP,UDP -> specific local ports: 9002, 9111

# 分布式应用系统的架构演变

Corba时代， 在1998年EJB1.0发布以后，就逐渐淡出历史舞台了

基于RMI/IIOP协议的EJB时代， 这个时代开始于1998年，到现在基本上已经划上了句号

不管是Corba，还是EJB，都有一些共同点：   
1) 通过专有的网络协议通讯   
2) 不能跨平台调用   
3) 通过分布式对象调用来实现分布式架构，换句话来说就是，分布式架构是绑定在面向对象的机制上的

基于SOAP协议的Web Services时代，个时代始于2001

web services有一些明显不同于Corba和EJB分布式对象架构的特征：   
1) 通过标准SOAP协议通讯，一般走HTTP通道   
2) 能够跨平台调用   
3) 通讯格式是xml文本，而不是二进制数据格式   
4) 通过RPC机制来实现分布式调用，而不是通过面向对象机制实现分布式调用

REST是专门为分布式调用设计的架构，在REST里面，分布式是通过对资源的操作来实现的，不是像EJB那样通过对象的方法调用来实现的。资源是一种抽象的概念，资源被映射到相应的一套URL规则上面了

分布式架构 协议 调用方式

-------------------------------------------------------

Corba架构 专有二进制协议 对象的CRUD操作

EJB架构 专有二进制协议 对象的CRUD操作

Web Services SOAP协议 RPC方式

REST HTTP协议 对资源的CRUD操作

--------------------------------------------------------

REST基础概念：

* 在REST中的一切都被认为是一种资源。
* 每个资源由URI标识。
* 使用统一的接口。处理资源使用POST，GET，PUT，DELETE操作类似创建，读取，更新和删除（CRUD）操作。
* 无状态。每个请求是一个独立的请求。从客户端到服务器的每个请求都必须包含所有必要的信息，以便于理解。
* 通信都是通过展现。例如XML，JSON

[SOA](http://www.cioage.com/soa/)的架构风格，它是基于企业业务架构的功能性分解，并且引入了两个高层次的抽象：企业业务服务和业务流程。企业业务服务代表的是现有IT能力（和企业的业务功能相一致）。业务流程编排业务服务，并定义业务的整体功能。

面向资源架构（ROA）的架构准则。ROA构建在资源这一概念之上；每个资源都是一个能够直接访问的分布式组件,可通过一个标准的、通用的接口来处理。所以，面向资源的架构（ROA）其最根本的还是一种基于资源的分解。

在最简单的情况下，服务可以被定义为一个自包含、独立开发、可部署、可管理和可维护的软件实现，它从整体上为企业提供特定的与业务相关的功能，并且在设计上是“可集成的”。

服务接口（尤其对某个给定的服务而言）定义服务功能，并且可由多种方式实现。存在两种基本的定义服务接口的方法——RPC风格和消息 （messaging）风格，RPC风格实现使用服务调用语义并且通过服务接口中的一组参数来定义。而消息风格的服务接口被有效地固定（本质上只需要进行 “执行”操作）使用XML文档作为输入和输出（这和GoF设计模式非常相似）。在这种情况下，服务语义是由输入和输出消息的语义来确定。

资源可以被定义为一个可直接访问的、独立开发的、可部署的、可管理的和可维护的软件构件，它支持特定的数据。一个资源就类似于一个对象，不过它是带有预定义（CRUD）接口语义的对象。实际上，一个资源就类似于一个对象，不过它是带有预定义（CRUD）接口语义的对象。

createResource——创建一个新的资源（以及相应的唯一标示）– PUT

getResourceRepresentation——获取资源信息– GET

deleteResource ——删除资源（可选地包括相关联的资源）– DELETE（只是引用的资源），POST（当需要删除相关联的资源时使用）

modifyResource——更改资源— POST

getMetaInforatmion——取得资源元数据信息—HEAD

资源通过两部分定义：资源URL和资源所提供的所有操作上定义的输入/输出参数。这和服务不同，服务的方法之间是完全独立，并且能够以独立端点（endpoints）的方式部署，而资源上的方法遵循OO语义，这意味着所有的方法（除createResource以外）都必须依附于底层的某个资源（同一个URL）。

如果把WS-\*比作是互联网世界的RPC，那么REST就是互联网世界的数据库管理系统（DBMS）……传统的基于[SOA](http://www.cioage.com/soa/" \t "_blank)的 集成表现了不同软件构件之间通过各种过程或方法进行交互。REST有效地将每个软件构件看作一组数据库表，而这些构件之间使用SELECT, INSERT, UPDATE和DELETE来通信。（或如你所想的使用GET, PUT, POST, DELETE）。那业务逻辑放在哪里呢？在存储过程中？不太对，其实在触发器中。

REST Web服务方法是指单纯使用REST技术作为通信手段来构建[SOA](http://www.cioage.com/soa/" \t "_blank)的一种方法

[SOA](http://www.cioage.com/soa/)P和REST最主要的区别在于REST是直接实现于HTTP协议之上，而[SOA](http://www.cioage.com/soa/" \t "_blank)P引人了一个抽象层（[SOA](http://www.cioage.com/soa/" \t "_blank)P消息传递）

REST是面向资源的，而每个资源之所以被称作为资源是因为他们能 够被识别和利用，这便是统一资源标识符（Universal Resource Identifier）被设计和如此命名的缘由。URI是网络上任何一个对象的名字，而它也仅仅是一个名字而已。URI就好比是资源的名字，它被用来标识、引用和查找资源，与现实世界中人名不同的是在网络上URI不会重复。

RPC是面向活动或方法的，无论这一方法是一个对象的成员方法还是全局 方法，我们习惯了通过客户端传递参数给他们然后获得返回值的模式，

对URI本身的“操作” 只存在两种：referencing和dereferencing，前者引用一个URI来指代一个资源，后者表示了通过URI来取回实际对象的命令，

REST定义了一个架构设计原则，其核心思想是服务的设计以系统资源为中心而不是以服务的功能为中心，包括资源的状态如何表示并如何通过HTTP传输给用不同语言编写的客户端

REST定义了一个架构设计原则，其核心思想是服务的设计以系统资源为中心而不是以服务的功能为中心，包括资源的状态如何表示并如何通过HTTP传输给用不同语言编写的客户端。

**当我们在设计Restful服务的时，应该遵循一下设计原则：**

**原则一： 使用HTTP的方法进行资源访问**

1) 使用HTTP POST方法去创建 资源

2) 使用HTTP GET方法去读取 资源

3) 使用HTTP PUT 方法去更新 资源

4) 使用HTTP DELETE方法去删除 资源

**原则二： 使用无状态/无会话的服务设计**

**原则三： 用目录结构风格的URL设计来表示资源**

用清晰的URL路径表示资源可以使客户端更容易理解和操作资源。URL可以被看作是一种自我解释的接口，不需要太多解释就可以让人明白该URL指向的是什么资源以及如何获得相关的资源。

**原则四： 使用XML或JSON来传输数据**

服务和请求的消息数据中包含了对于资源的属性的描述，服务应该采取结构良好并且易于阅读的方式来描述资源。资源可能是数据库中的某个记录集合或者是一个具体的记录，可以是文档，甚至可以是数据中心的服务器

代表性状态传输 (REST) 是一种用于分布式超媒体系统（如万维网）的体系结构样式。REST 风格的体系结构的核心概念是由统一资源标识符 (URI) 标识的资源。可以使用标准接口（如 HTTP）来处理这些资源以及使用这些资源的表示形式来交换信息。

Web Services 是一种基于组件的软件平台,是面向服务的Internet 应用。Web Services 是应用于Internet 的,而不是限于局域网或试验环境。这要求提出的Web Services 框架必须适用于现有的Internet 软件和硬件环境,即服务的提供者所提供的服务必须具有跨平台、跨语言的特性。其次,Web Services 所提供的服务不仅是向人,更需服务于其它应用系统。现有的Web网站也可以认为是面向服务的,但这种服务仅仅可以提供给人使用(只有人类才可以读懂浏览器 下载的页面) 。而新一代的Web Services 所提供的服务应能被机器所读懂,例如其它应用程序及移动设备中的软件系统。这样,我们可以看出,Web Services 的发展方向实际上是构造一个在现有Internet 技术上的分布计算系统。

　SOAP 是Web services 的通信协议。SOAP是一种简单的、轻量级的基于XML 的机制，用于在网络应用程序之间进行结构化数据交换。SOAP包括三部分:一个定义描述消息内容的框架的信封，一组表示应用程序定义的数据类型实例的编码 规则，以及表示远程过程调用和响应的约定。

WSDL表示WEB服务说明语言。WSDL文件是一个XML 文档，用于说明一组SOAP消息以及如何交换这些消息。

UDDI(统一描述发现和集成) 提供一种发布和查找服务描述的方法。UDDI 数据实体提供对定义业务和服务信息的支持。WSDL 中定义的服务描述信息是UDDI注册中心信息的补充。

Web Services 的技术主要建立在XML 的规范之上,这保证了这一体系结构的平台无关性、语言无关性和人机交互性能。

# Authentication and Authorization

Web 应用的安全性包括**用户认证（Authentication）和用户授权（Authorization）**两个部分。

用户认证指的是验证某个用户是否为系统中的合法主体，也就是说用户能否访问该系统。

用户授权指的是验证某个用户是否有权限执行某个操作。在一个系统中，不同用户所具有的权限是不同的。比如对一个资源来说，有的用户只能进行读取，而有的用户可以进行修改。一般来说，**系统会为不同的用户分配不同的角色，而每个角色则对应一系列的权限**

认证手段一般是保存到数据库表的用户名/密码，usb key，ldap等。一般情况下，我们都使用用户名/密码的

身份验证：

连同用户证书一起发送请求到指定URL，如果请求成功，浏览器就返回cookie,浏览器将稍后自动发送该cookie验证用户。

## AD LDS Server

目录服务其实也是一种数据库系统，只是这种数据库是一种树形结构，而不是通常使用的关系数据库。目录服务与关系数据库之间的主要区别在于：二者都允许对存储数据进行访问，只是目录主要用于读取，其查询的效率很高，而关系数据库则是为读写而设计的

访问数据库：

Start > Administrative Tools > ADSI Edit > Connection Settings:

Computer:

Select or Type a domain or server: qq0006cr450w3sw.code1.emi.philips.com:389

Connection Point:

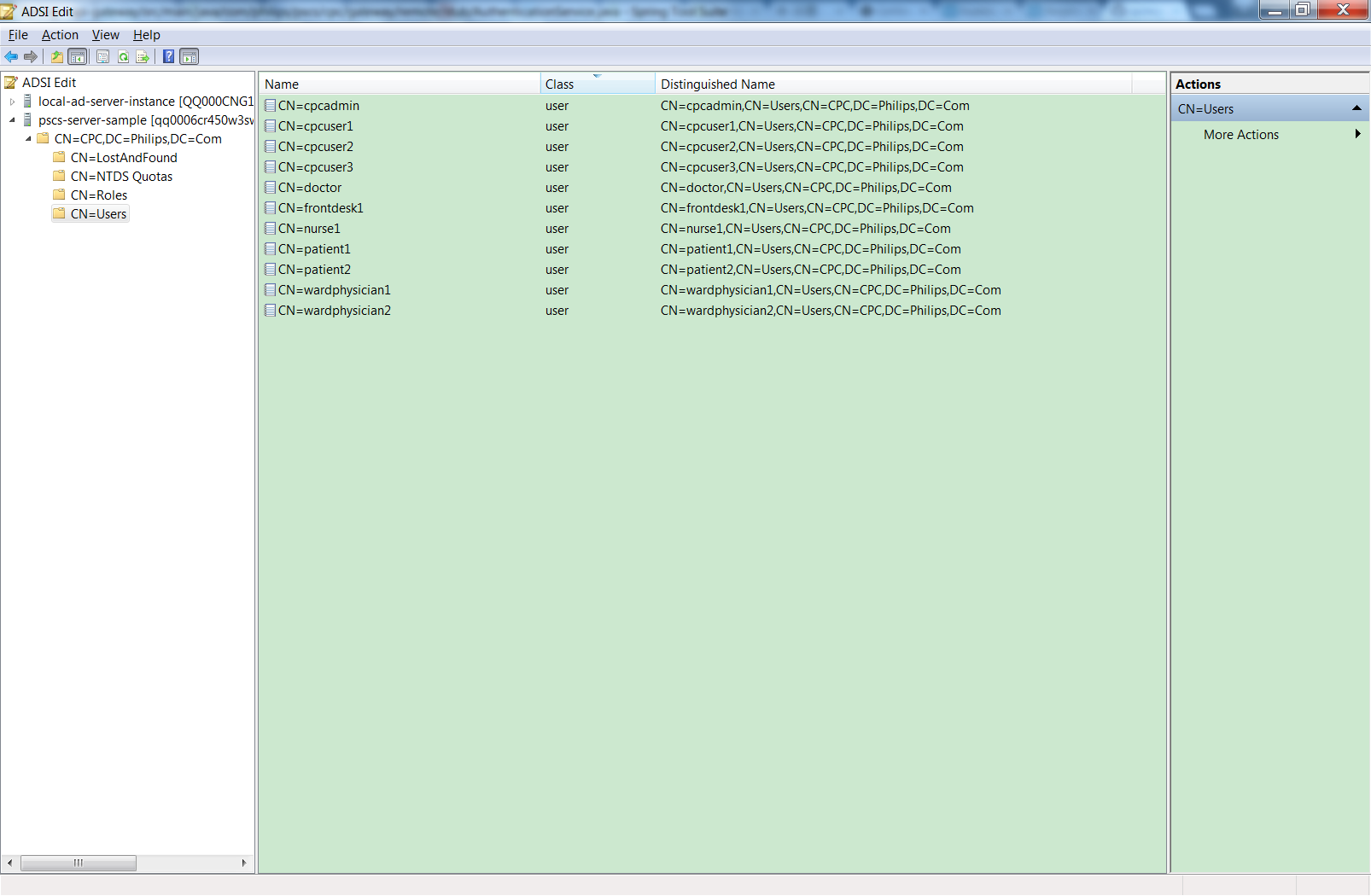
Select or type a Distinguished Name or Naming Context:

CN=CPC,DC=Philips,DC=Com

Name: name instance you want

则自动生成Path: LDAP://qq0006cr450w3sw.code1.emi.philips.com:389/CN=CPC,DC=Philips,DC=Com

Click ok -> ADSI Edit界面



1. 添加用户信息

Right click CN=CPC,DC=Philips,DC=Com -> new Objects… -> select container -> value: Users

Right click -> new Objects…-> select user -> value: username1

Double click CN= username1-> select one attribute -> click Edit -> edit value

Note: right click -> CN=username1 -> Reset Password -> pass@123

密码是不显示在Attribute Editor界面中，同样用代码也无法查询到。用户密码比对一般是在AD Server内部执行。发送密码到AD Server保存时，需要用md5进行加密

1. 添加角色信息

Right click CN=CPC,DC=Philips,DC=Com -> new Objects…-> select container -> value: Roles

Right click -> new Objects… -> select group -> value: WardPhysician

Double click CN=WardPhysician -> double click member -> Add DN… -> CN=wardphysician2,CN=Users,CN=CPC,DC=Philips,DC=Com

角色可以有很多用户，用户可以赋于多种角色，简单地说，角色和用户是多对多关系

## 邮件发送

SMTP（Simple Mail Transfer Protocol）即简单邮件传输协议,它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式，属于TCP/IP协议簇。SMTP服务器则是遵循SMTP协议的发送邮件服务器，用来发送或中转发出的电子邮件。

跟大多数应用层协议一样，SMTP也存在两个 端：在发信人的邮件服务器上执行的客户端和在收信人的邮件服务器上执行的服务器端。SMTP的客户端和服务器端同时运行在每个邮件服务器上。当一个邮件服务器在向其他邮件服务器发送邮件消息时，它是作为SMTP客户在运行。

SMTP通常有两种工作模式：发送SMTP和接收SMTP。具体工作方式为：发送SMTP在接到用户的邮件请求后，判断此邮件是否为本地邮件，若是直接投送到用户的邮箱，否则向dns查询远端邮件服务器的MX纪录，并建立与远端接收SMTP之间的一个双向传送通道。具体过程如下：

（1）建立TCP连接。

（2）客户端向服务器发送HELO命令以标识发件人自己的身份，然后客户端发送MAIL命令。

（3）服务器端以OK作为响应，表示准备接收。

（4）客户端发送RCPT命令。

（5）服务器端表示是否愿意为收件人接收邮件。

（6）协商结束，发送邮件，用命令DATA发送输入内容。

（7）结束此次发送，用QUIT命令退出。

SMTP服务器基于DNS中的邮件交换（MX）记录路由电子邮件。电子邮件系统发邮件时是根据收信人的地址后缀来定位邮件服务器的。SMTP通过用户代理程序（UA）完成邮件的编辑、收取和阅读等功能；通过邮件传输代理程序（MTA）将邮件传送到目的地。

# Domain-specific language

## Drools

规则引擎一般由3部分组成：规则库(Knowledge base)+Working Memory(Fact base)+推理机(规则引擎)。

使用 Drools 规则引擎实现业务逻辑

使用声明性编程方法编写程序的业务逻辑

实现 J2EE 或 J2SE 应用程序中业务逻辑最常见的方法是编写 Java 代码来实现需求文档的规则和逻辑。在大多数情况下，该代码的错综复杂性使得维护和更新应用程序的业务逻辑成为一项令人畏惧的任务，任何更改，不管多么简单，仍然会产生重编译和重部署成本。

将规则引擎看作实现复杂业务逻辑的框架。大多数规则引擎允许您使用声明性编程来表达对于某些给定信息或知识有效的结果

有多个规则引擎可供使用，其中包括商业和开放源码选择。商业规则引擎通常允许使用专用的类似英语的语言来表达规则。其他规则引擎允许使用脚本语言（比如 Groovy 或 Python）编写规则

Eclipse IDE（Versions 3.2 和 3.3）的一个 Drools 插件。我强烈建议您通过这个插件来使用 Drools。它可以简化使用 Drools 的项目开发，并且可以提高生产率。例如，该插件会检查规则文件是否有语法错误，并提供代码完成功能。它还使您可以调试规则文件，将调试时间从数小时减少到几分钟。您可以在规则文件中添加断点，以便在规则执行期间的特定时刻检查对象的状态。这使您可以获得关于规则引擎在特定时刻所处理的知识（knowledge）（在本文的后面您将熟悉这个术语）的信息。

何时使用规则引擎？

并非所有应用程序都应使用规则引擎。如果业务逻辑代码包括很多 if-else 语句，则应考虑使用一个规则引擎。维护复杂的 Boolean 逻辑可能是非常困难的任务，而规则引擎可以帮助您组织该逻辑。当您可以使用声明方法而非命令编程语言表达逻辑时，变化引入错误的可能性会大大降低。

如果代码变化可能导致大量的财政损失，则也应考虑规则引擎。许多组织在将已编译代码部署到托管环境中时具有严格的规则。例如，如果需要修改 Java 类中的逻辑，在更改进入生产环境之前，将会经历一个冗长乏味的过程：

必须重新编译应用程序代码。

在测试中转环境中删除代码。

由数据质量审核员检查代码。

由托管环境架构师批准更改。

计划代码部署。

即使对一行代码的简单更改也可能花费组织的几千美元。如果需要遵循这些严格规则并且发现您频繁更改业务逻辑代码，则非常有必要考虑使用规则引擎。

对客户的了解也是该决策的一个因素。尽管您使用的是一个简单的需求集合，只需 Java 代码中的简单实现，但是您可能从上一个项目得知，您的客户具有在开发周期期间甚至部署之后添加和更改业务逻辑需求的倾向（以及财政和政治资源）。如果从一开始就选择使用规则引擎，您可能会过得舒服一些。

# 测试

## 压力测试

JMeter 是 Apache 基金会 Jakarta 上的一个纯 Java 开源项目，起初用于基于 Web 的压力测试（pressure test），后来其应用范围逐渐扩展到对文件传输 FTP, 大型数据库（JDBC 方式），脚本程序（CGI, Perl 等），Web Services，Java 应用系统等方面的测试。JMeter 本身主要用于性能测试，如系统压力等。除此之外，JMeter 能够对应用系统做功能测试和回归测试，并且能够通过使用带有断言的脚本程序来验证系统然后返回用户期望的结果

消息传递协议：SOAP

SOAP（Simple Object Access Protocol）称为简单对象访问协议， 是 W3C 定义的一种标准消息传递协议，而它通常被认为是 Web Services 的事实标准。SOAP 是在去中心化（Decentralized）分布式（Distributed）环境中用来信息交换的一个轻量级协议。

SOAP 使用 RPC（远程过程调用）和消息传递来建立通信服务，SOAP RPC 定义了用于表示远程过程调用和应答的协议。SOAP 协议本身仅仅定义了消息的交换结构，它可以和许多现存因特网协议结合在一起使用，其中包括超文本传输协议（ HTTP），Java 消息服务（JMS）

HTTP（超文本传送协议）是属于应用层的面向对象的协议，是万维网 (WWW) 的基础，由于其简单快速、灵活、无连接、无状态的方式，适用于分布式网络信息系统。SOAP Over HTTP 应用就是指的是遵守 SOAP 编码规则的 HTTP 请求 / 响应，我们可以用简单的公式来对此作一个描述：HTTP + XML = SOAP。

JMeter 也同样提供了两种 Sampler 分别建立对这两种服务的调用：Web Services (SOAP) Request 和 JMS Point-to-Point

# Office Note

(modify the document with record)

Review -> Track Changes

Review -> New Comments

Review -> Accept

分布式与集群

分布式是指将不同的业务分布在不同的地方；而集群指的是将几台服务器集中在一起，实现同一业务。分布式中的每一个节点，都可以做集群。

Linux集群

主要分成三大类( 高可用集群， 负载均衡集群，科学计算集群)

负载均衡集群(Load Balance Cluster)

负载均衡系统：集群中所有的节点都处于活动状态，它们分摊系统的工作负载。一般Web服务器集群、数据库集群和应用服务器集群都属于这种类型。

负载均衡集群一般用于相应网络请求的网页服务器，数据库服务器。这种集群可以在接到请求时，检查接受请求较少，不繁忙的服务器，并把请求转到这些服务器上。从检查其他服务器状态这一点上看，负载均衡和容错集群很接近，不同之处是数量上更多。