工程技术规范

（STPS）

东软集团

商用软件事业部

版本V1.0

**变更控制**

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 时间 | 修改人 | 内容 |
| V1.0 |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

[1. 总体说明 7](#_Toc14688293)

[1.1 目的和意义 7](#_Toc14688294)

[1.2 术语表 7](#_Toc14688295)

[1.3 符号说明 7](#_Toc14688296)

[1.4 技术过程说明 8](#_Toc14688297)

[1.5 与其他标准的关系 10](#_Toc14688298)

[1.6 常见问题 11](#_Toc14688299)

[2. 架构设计(Architecture Definition) 11](#_Toc14688300)

[3. 实现设计(Design Definition)-全局设计域 11](#_Toc14688301)

[3.1 分层/模块策略 11](#_Toc14688302)

[3.2 工程结构设计 11](#_Toc14688303)

[3.3 WebApi设计方案 11](#_Toc14688304)

[3.4 性能设计 11](#_Toc14688305)

[3.5 国际化 11](#_Toc14688306)

[4. 实现设计(Design Definition)-公用模块域 12](#_Toc14688307)

[4.1 弹出消息(Messages) 12](#_Toc14688308)

[4.1.1 定义 12](#_Toc14688309)

[4.1.2 适用范围 12](#_Toc14688310)

[4.1.3 术语表 13](#_Toc14688311)

[4.1.4 区别与联系 13](#_Toc14688312)

[4.1.5 弹出消息的设计原则 13](#_Toc14688313)

[4.1.6 弹出消息参考设计 13](#_Toc14688314)

[4.2 数据验证 19](#_Toc14688315)

[4.3 异常处理 19](#_Toc14688316)

[4.3.1 定义 19](#_Toc14688317)

[4.3.2 适用范围 19](#_Toc14688318)

[4.3.3 术语表 19](#_Toc14688319)

[4.3.4 区别与联系 20](#_Toc14688320)

[4.3.5 设计原则 20](#_Toc14688321)

[4.3.6 参考设计 21](#_Toc14688322)

[4.4 日志处理 21](#_Toc14688323)

[4.4.1 定义 21](#_Toc14688324)

[4.4.2 适用范围 21](#_Toc14688325)

[4.4.3 术语表 21](#_Toc14688326)

[4.4.4 区别与联系 21](#_Toc14688327)

[4.4.5 设计原则 21](#_Toc14688328)

[4.4.6 参考设计 21](#_Toc14688329)

[4.5 配置文件 21](#_Toc14688330)

[5. 实现设计(Design Definition)-数据访问域 22](#_Toc14688331)

[5.1 排他设计 22](#_Toc14688332)

[5.2 数据库连接策略 22](#_Toc14688333)

[5.3 事务处理 22](#_Toc14688334)

[6. 实现设计(Design Definition)-安全域 22](#_Toc14688335)

[6.1 用户认证方案 22](#_Toc14688336)

[6.2 角色权限方案 22](#_Toc14688337)

[6.3 敏感信息加密方案 22](#_Toc14688338)

[6.4 恶意攻击预防 22](#_Toc14688339)

[7. 系统分析(System Analysis) 23](#_Toc14688340)

[7.1 关键指标验证 23](#_Toc14688341)

[7.1.1 并发量 23](#_Toc14688342)

[7.1.2 数据库处理 23](#_Toc14688343)

[7.1.3 响应时间 23](#_Toc14688344)

[7.1.4 批处理时间 23](#_Toc14688345)

[7.2 外部系统调用与数据交换 23](#_Toc14688346)

[7.2.1 认证机制 23](#_Toc14688347)

[7.2.2 时序机制 23](#_Toc14688348)

[7.2.3 数据格式 23](#_Toc14688349)

[7.2.4 数据交换文件 23](#_Toc14688350)

[8. 程序构建(Implementation) 24](#_Toc14688351)

[8.1 开发环境搭建 24](#_Toc14688352)

[8.2 引用包与依赖管理 24](#_Toc14688353)

[8.3 stub与mock设计 24](#_Toc14688354)

[8.4 命名规范 24](#_Toc14688355)

[8.5 测试数据组织 24](#_Toc14688356)

[8.6 测试环境组织 24](#_Toc14688357)

[9. 模块集成(Integration) 25](#_Toc14688358)

[9.1 集成策略 25](#_Toc14688359)

[9.2 测试数据组织 25](#_Toc14688360)

[9.3 测试环境组织 25](#_Toc14688361)

[9.4 版本发布策略 25](#_Toc14688362)

[10. 系统测试(Verification) 25](#_Toc14688363)

[10.1 测试策略 25](#_Toc14688364)

[10.2 测试数据组织 25](#_Toc14688365)

[10.3 测试环境组织 25](#_Toc14688366)

[11. 移植项目专题 26](#_Toc14688367)

[11.1 移植策略与原则 26](#_Toc14688368)

[11.2 原系统分析 26](#_Toc14688369)

[11.3 新系统架构设计 26](#_Toc14688370)

[11.4 移植工程设计 26](#_Toc14688371)

[11.5 集成策略 26](#_Toc14688372)

[11.6 对比测试策略 26](#_Toc14688373)

[11.7 工具运用 26](#_Toc14688374)

# 总体说明

## 目的和意义

本文档是针对日软件外包业务，结合了商用软件事业部的长期软件工程实践，参考国际标准，广泛吸收行业内的先进实践，总结出来的用于指导项目技术过程的标准体系。

编写本文档的主要目的有以下几点：

1. 为项目的技术工作展开提供参考框架和依据
2. 保障项目技术过程的有效性，提高效率
3. 为组织层面共享技术实践提供索引
4. 作为组织级工程技术能力的载体

## 术语表

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 术语 | 英文 | 解释 |
| 1 | 干系人 | stakeholder | 也叫利益相关者，项目干系人是参与该项目工作的个体和组织，或由于项目的实施与项目的成功，其利益会直接或间接地受到正面或负面影响的个人和组织。 |
| 2 |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## 符号说明

推荐 ：推荐大多数项目采用的情形

禁止 ：严禁在项目中出现的情形

强制 ：无特殊理由，必须按照标准规范执行的情形

## 技术过程说明

项目的技术过程定义了一系列活动，这些活动使得组织或者项目能够从技术决策和行动中获得收益的最大化，并且避免技术风险。根据ISO/IEC 12207:2017，项目的技术过程包括：商业目标分析、干系人诉求和需求定义、系统/软件需求定义、架构设计、实现设计、实现分析、程序构建、模块集成、系统测试、移交、验收测试、上线运行、维护、废止。

**注意：**ISO/IEC 12207:2017中定义的技术过程，与传统意义上瀑布模型中的工程阶段没有关联，在时间顺序上这些过程可能在不同的时间点上反复进行。这些技术过程不依赖于任何软件开发模型（software development life cycle）而存在，所以，技术过程对包括敏捷模型的各种软件开发模型，都是适用的。

* + **商业目标分析(Business or Mission Analysis process)**

主要目的：定义商业目标、任务所要解决的问题，或者商业机会。描绘解决方案的空间，即候选方案。确定能够解决问题或者从商机中获益的潜在解决方案。

注：相当于日本软件行业的“企划”

* + **干系人诉求和需求定义(Stakeholder Needs and Requirements Definition process)**

主要目的：定义干系人对系统的需求，该系统能够在特定的环境下为使用者和其他干系人的提供他们所需的能力。

注：相当于日本软件行业中的“要件定义”

* + **系统/软件需求定义(System/Software requirements definition process)**

主要目的：把干系人、用户视角所期待的能力，转换成技术视角的解决方案，这个解决方案能够在运用层面上实现用户的需求。

注：相当于日本软件行业中的“基本设计”

* + **架构设计(Architecture Definition process)**

主要目的：创建候选系统架构方案，选择一个或者多个能够满足干系人关键诉求和系统需求的候选方案，用统一的一系列视图描述架构设计方案。

架构设计的重点是选择、定义、描述方案，实现设计的重点是具体描述怎样实现预设方案。

软件架构：软件架构是一系列高层次的决策，这些决策能够满足干系人的关键诉求。比如：客户要求系统达到99.99%的可用性，那么在架构设计中就要有相应的高可用方案，比如从双活、主备等集群方案中选择最适合当前项目的方案。

* + **实现设计(Design Definition process)**

主要目的：提供系统和其组成元素的充分细节和信息，从而能够基于这些信息来实现架构实体，这些实体来源于架构定义中的模型和视图。

* + **系统分析(System Analysis process)**

主要目的：作为一种对项目的技术理解，提供精确缜密的基础数据和信息，从而为各个过程阶段提供决策依据。

实现分析可以使用各种手段，包括逻辑上的和物理上的。比如，可以开发模拟程序去验证性能，基于测试结果推算系统是否满足性能指标。

* + **程序构建(Implementation process)**

主要目的：具体实现系统的各个组成单元。

* + **模块集成(Integration process)**

主要目的：在满足系统/软件需求架构和设计要求的前提下，把系统的各个单元组合成现实的系统（产品或者服务）。

* + **系统测试(Verification process)**

主要目的：提供客观的证据，从而证明系统或者系统单元能够完全满足指定的需求和特性。

* + **移交(Transition process)**

主要目的：在生产环境中建立系统能力，满足干系人需求。

* + **验收测试(Validation process)**

主要目的：提供客观证据，从而验证系统在运用过程中，能够满足最初的商业目标或者业务目标、干系人的需求，以及在预定环境中实现预定用途。

* + **上线运行(Operation process)**

主要目的：使用系统交付服务。

* + **维护(Maintenance process)**

主要目的：维持系统提供服务的能力。

* + **废止(Disposal process)**

主要目的：结束特定的具有预定功能的系统单元或者系统系统的运行，恰当地处理拟替换或者废止的系统单元，识别和废止有关的关键需求。（遵循协议、遵循组织策略，考虑环境、法律、安全、保密等方面）

## 与其他标准的关系

本文对于技术过程的定义，以ISO/IEC 12207 2017为准，与各国际客户的工程标准对应关系如下表：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 序号 | ISO定义 | 东芝标准 | Unisys标准 |  |  |
| 1 | 商业目标分析 | SP | - |  |  |
| 2 | 干系人诉求和需求定义 | RD | 要件定義 |  |  |
| 3 | 系统/软件需求定义 | BD | 論理設計 |  |  |
| 4 | 架构设计 | RD～BD | 論理設計～物理設計・実装 |  |  |
| 5 | 实现设计 | SD | 物理設計・実装 |  |  |
| 6 | 实现分析 | - | - |  |  |
| 7 | 程序构建 | DD～PT | 物理設計・実装～単体テスト |  |  |
| 8 | 模块集成 | IT | - |  |  |
| 9 | 系统测试 | ST | - |  |  |
| 10 | 移交 | - | - |  |  |
| 11 | 验收测试 | CT | - |  |  |
| 12 | 上线运行 | - | - |  |  |
| 13 | 维护 | - | - |  |  |
| 14 | 废止 | - | - |  |  |

## 常见问题

# 架构设计(Architecture Definition)

预留，第二期内容。

# 实现设计(Design Definition)-全局设计域

## 分层/模块策略

## 工程结构设计

## WebApi设计方案

### 什么是WebApi

WebApi就是一个Web系统，通过访问URI可以与服务器完成信息交互，或者获得存在在服务器的数据信息。这样调用者通过程序可以进行访问后既可以机械的使用这些数据。

设计良好的Web API应该基本满足

1. 平台独立性，不管API的内部实现方式如何，理论上任何客户端都应该调用该API。这就需要使用标准协议并创建一种机制，试客户端和Web服务能够就交互数据的格式达成一致目标。
2. 服务演变，Web API应能在不影响客户端应用程序的情况下改进和添加功能。

### 设计原则

REST是REpresentational State Transfer的缩写（一般中文翻译为表述性状态转移），REST 是一种体系结构，而 HTTP 是一种包含了 REST 架构属性的协议，为了便于理解，我们把它的首字母拆分成不同的几个部分：

表述性（REpresentational）： REST 资源实际上可以用各种形式来进行表述，包括 XML、JSON 甚至 HTML——最适合资源使用者的任意形式；

状态（State）： 当使用 REST 的时候，我们更关注资源的状态而不是对资源采取的行为；

转义（Transfer）： REST 涉及到转移资源数据，它以某种表述性形式从一个应用转移到另一个应用。

简单地说，REST 就是将资源的状态以适合客户端或服务端的形式从服务端转移到客户端（或者反过来）。在 REST 中，资源通过 URL 进行识别和定位，然后通过行为(即 HTTP 方法)来定义 REST 来完成怎样的功能。

我们再设计api的时候，需要参照和借鉴REST的设计思想

### 围绕资源组织API

一种非常简单的API 设计方法是编制一套数据访问机制，它能够直接操作在线服务所用的数据库及其数据表里的信息

但如果仅通过封装SQL 语句来进行API 的设计，那么开发出来的API 会非常难用。因为对于这样的API，如果人们不理解其内部数据如何存放、数据之间存在怎样的关系，就无法正确使用。并且这样的设计还会将服务内部数据的存储结构公开，从安全角度来说存在很大的风险。因此，API 必须在更高的层次来描述相关的功能。那我们该如何设计API 呢？首先，需要知道用户会如何使用对外公开的API，仔细思考用户的用例场景。根据项目的不同可能有多种方式调用API(例如通过h5页面，移动页面等等)，在基于用户的场景基础上，确认API需要的功能。之后围绕资源组织API。如果可能，资源 URI 应基于名词（资源）而不是动词（对资源执行的操作）。

https://adventure-works.com/orders // 好的设计

https://adventure-works.com/create-order // 差的设计

在 URI 中采用一致的命名约定。 一般而言，有效的做法是对引用集合的 URI 使用复数名词。 最好是将集合和项的 URI 组织成层次结构。 例如，/customers 是客户集合的路径，/customers/5 是 ID 为 5 的客户的路径。 这种方法有助于使 Web API 保持直观。 此外，有许多 Web API 框架可以基于参数化 URI 路径来路由请求，因此，你可以对路径 /customers/{id} 定义路由。

还需要考虑不同类型的资源之间的关系，以及如何公开这些关联。 例如，/customers/5/orders 可以表示客户 5 的所有订单。 我们还可以改变思维方向，使用类似于 /orders/99/customer 的 URI 来表示从订单到客户的关联。 但是，过度扩展此模型可能会变得难以实现。 在设计API的URI的时候需要进行权衡，基本的思路尽量的简洁

### 设计最佳实践(check list)

1 . 短小便于输入的URI

“短小便于输入”意味着URI 简单易记，而冗长的URI 往往就会混有无用、重复的内容。让我们来看一下下面这个URI 端点的例子。

http://api.example.com/service/api/search

该URI 中包含了“api”“search”等单词，可以知道这是一个用于检索某种信息的API。但因为主机名和路径里都包含了“api”，所以显得有点重复。另外，该URI中还包含了“service”这类表示雷同概念的单词。实际上，该URI 和下面这个简短的URI 所包含的信息基本没有什么差别。

http://api.example.com/search

通过该URI 我们也能得知这是一个用于检索某种信息的API（如果不是用来检索信息的API，问题就严重了）。在表示的信息量相同的情况下，使用短小、简单的表述方式更易于理解和记忆，并能减少输入时的错误。

2 . 人可以读懂的URI

“人可以读懂的URI”是指，像上面提到的用来检索信息的API 的URI 那样，一看到该URI，即使没有其他提示，也能理解其用途。

例如，以下URI 就是一个意思不明确的URI。

http://api.example.com/sv/u/

因为该URI 里包含了“api”一词，所以可以得知它是某API 的URI，但“sv”和“u”表示什么意思就不得而知了。或许是某个词的缩写，“u”可能是指user，“sv”可能是指service，但这些都无法确定。或许该URI 的设计人员想把URI 设计得短小精悍，但却导致URI 本身难以理解。

为了避免设计出这样难以理解的URI，首先就要做到不轻易使用缩写形式，例如将products 缩写为prod，把week 缩写为wk 等。虽然有人认为这些缩写形式对于母语是英语的人而言非常常见，但即使这样，也不建议过度使用省略和缩写形式。因为使用API 的开发人员的母语未必是英语 。

另外，即使看起来是同一个缩写形式，也存在细微的差别，比如表示国家名称的“jp”“jpn”等。关于国家代码，可以参考已有的ISO 3166 国际标准。使用标准化的“代码”体系来标识，比使用其他符号更易于理解。类似这样的标准化的代码体系还包括表示语言的ISO 639、表示航空公司及机场的代码体系（如日本航空为JL、羽田机场为HND）等。

要设计容易理解的URI，第2 个要点就是要使用API 里常用的英语单词。之所以建议使用英语单词，是因为使用全世界通用的英语来描述可以使API 更易于理解。以电子商务网站里获取某种商品信息的API 为例，其URI 如下所示。

http://api.example.com/products/12345

http://api.example.com/productos/12345

http://api.example.com/seihin/12345

第1 个URI 使用了英语单词“products”，第2 个URI 使用了西班牙语，第3 个URI 则用了日语罗马字母。这3 个URI 中哪一个最容易理解不言而喻。西班牙语同英语相近，容易混淆，大多数不懂西班牙语的人会觉得这个URI 的productos 一词多输入了一个o，也许很多人还会误用“products”那个URI 进行访问。而使用日语罗马字母的话，因为拼写完全不同，所以一眼看去完全不知所云，不懂日语的人根本无法猜测该URI 的含义，即使是日本人，对于原本用汉字描述的词汇改用罗马字母显示的情况，也同样需要花费一些时间来理解。

要设计容易理解的URI，第3 个要点就是要尽可能地避免拼写错误。对于母语非英语的人而言，这一点尤其应引起注意。所以一旦URI 出现拼写错误，就会让API 用户难以判断究竟是API 出现了拼写错误还是API 的文档出现了拼写错误，从而导致用户必须在开发过程中逐一确认，非常不便。

易于理解的URI 还可以减轻用户编写访问API 的代码时的负担。因为如果从URI 就能知道该API 的用途，那么开发人员在阅读访问该API 的代码时，就可以不用每次都去翻阅API 的相关文档，从而大幅提高了工作效率。另外，这也会减少开发人员因URI 难以理解而访问了错误的URI 进而访问了错误的API 的情形。

3 . 全部使用小写字母的URI

不要使用如下例所示的大小写混用的URI，一般建议全部使用小写字母的形式。

http://api.example.com/Users/12345

http://example.com/API/getUserName

大小写字母混用会造成API 难以理解，容易让人搞错。因此需要统一为全部大写或全部小写，一般标准的做法是全部使用小写。

4 . 修改方便的URI

“修改方便的URI”指的是能将某个URI 非常容易地修改为另一个URI。假设我们需要获取某种商品（item，根据API类型的不同而发生变化），该API 的端点如下所示。

http://api.example.com/v1/items/12346

从以上URI 能直观地看出该商品ID 为123456，并且可以猜测只要修改这一ID，就能访问到其他商品的信息。

该URI 的结构应在API 文档里明确记录下来。不过，如果认为只要在文档里进行了详细的说明，即使URI 设计得难以理解也丝毫没有影响的话就大错特错了。因为按以往的经验来看，开发人员往往不会仔细阅读文档，他们会马上进入埋头开发的状态。从API 用户的角度而言，如果开发过程中还需要不断地去翻阅文档，那么这样的API 显然是增加了开发人员的负担。

5 . 不会暴露服务器端架构的URI

“服务器端架构”信息包括使用了怎样的服务软件、使用了哪种开发语言来实现，以及服务器端的目录和系统结构等。假设客户端访问API 获取信息时需要访问以下端点。

http://api.example.com/cgi-bin/get\_user.php?user=100

从以上端点就可以知道该API 可能是用PHP 语言编写并以CGI 的方式运行。这些信息对API 用户来说显然是多余的。因为不管API 是用PHP 语言编写的还是用COBOL 语言编写的，对API 用户而言并没有什么区别。从另一方面来说，也有可能会有人想要了解这些信息，尤其是那些企图利用服务器漏洞实施恶意攻击的黑客。例如CGI 版本的PHP 的脆弱性已广为人知，2012 年人们发现利用该安全漏洞可以显示源代码并执行任意代码，这在当时引起了广泛的关注。以上URI 很容易暴露服务器端的架构信息，进而增加服务器端遭受攻击的可能性。

Web 应用里也同样无需在URI 中体现服务器端的架构和目录结构。对Web API而言，URI 理应体现功能、数据结构和含义，而不是服务器端是如何运作的等信息。

6 . 规则统一的URI

如果使用如下所示的规则统一的URI，便会很容易理解。

❖ 获取好友信息

http://api.example.com/friends/100

❖ 发送消息

<http://api.example.com/friends/100/message>

7. URL中的内容符合规范

使用RFC3986（<http://tools.ietf.org/html/rfc3986>)规范

建议只使用下面的字符

英文字（小文字）(a-z)/ 数字(0-9) / "-" / "." / "\_" / "~"

### Http的方法

在设计API 的访问方法时，必须同时考虑到这两个因素。HTTP 方法是进行HTTP 访问时指定的操作，包括了著名的GET/POST 操作等。开发Web 应用时，开发人员会在表单（Form）的method 属性中指定HTTP 方法选项，说到这里大家可能会有所理解。如下例所示，HTTP 方法会添加在HTTP 请求首部的第一行开头发送给服务器。

GET /v1/users/123 HTTP/1.1

Host: api.example.com

URI 和HTTP 方法之间的关系可以认为是操作对象和操作方法的关系。如果把URI 当作API（HTTP）的“操作对象= 资源”，HTTP 方法则表示“进行怎样的操作”。URI 里的R 表示“Resource”，即“资源”的意思，用于描述某种具体的数据信息。Web 页面的情况下，Web 页面所包含的内容就是一种资源；API 的情况下，可以通过端点获取的数据信息也是一种资源。HTTP 方 法所表示的就是对该资源进行怎样的操作，其中包括获取操作、修改操作、删除操作等。

通过用不同的方法访问一个URI 端点，不但可以获取信息，还能修改信息、删除信息等。因此我们可以将资源和对资源进行怎样的操作分开处理。这么做和HTTP 的设计思想也相吻合，Web API 中遵循这样的思想进行设计的方式也正成为主流。

开发Web 应用时，一种普遍的做法是通过GET 方法来获取服务器端的信息，而用POST 方法修改服务器端的信息。Web 页面里使用某元素A 的普通链接，可以视作使用GET 方法进行的访问。另外，在使用表单的情况下，可以选择POST 方法和GET 方法。

HTTP 协议中定义了更多的HTTP 方法（常用）。具体如下

|  |  |
| --- | --- |
| 方法名 | 说明 |
| GET | 读取资源，幂等 |
| POST | 新建或者更新资源，非幂等 |
| PUT | 更新资源 幂等 |
| DELETE | 删除资源 |
| PATCH | 更新部分资源 |

注意事项：

1. 对于Get方法的参数有最大限制，不能超过2000byte，超过的场合下使用POST方法
2. 一次请求大量数据的场合下，需要考虑通过业务限制数据量的大小，例如采用分页方案，客户端进行多次加载
3. PDF，Excel下载的场合下，一般按照Get方法进行设计

具体事例如下

GET /zoos：列出所有动物园

POST /zoos：新建一个动物园

GET /zoos/ID：获取某个指定动物园的信息

PUT /zoos/ID：更新某个指定动物园的信息（提供该动物园的全部信息）

PATCH /zoos/ID：更新某个指定动物园的信息（提供该动物园的部分信息）

DELETE /zoos/ID：删除某个动物园

GET /zoos/ID/animals：列出某个指定动物园的所有动物

DELETE /zoos/ID/animals/ID：删除某个指定动物园的指定动物

如上同样是

### API常用的http的状态码

在设计web api的时候，客户端的每一次请求，服务器都必须给出回应。回应包括 HTTP 状态码和数据两部分。

常用的WEBAPI的状态码如下

|  |  |
| --- | --- |
| 方法名 | 说明 |
| 2xx: Success | 客户端的请求被成功地接收 |
| 3xx: Redirection | 客户端必须有一些附加的动作才能完成它们的请求 |
| 4xx: Client Error | 此类错误应该由客户端负责 |
| 5xx: Server Error | 服务器对此类错误负责 |

200状态码表示操作成功，但是不同的方法可以返回更精确的状态码（一版项目推荐到200即可以满足需求）。其他WEB API可以选用的状态码

201 资源被成功的创建（针对POST请求）

202 Accepted - 服务器接受了请求，但是还未处理，响应中应该包含相应的指示信息，告诉客户端该去哪里查询关于本次请求的信息。

204 No Content - 对不会返回响应体的成功请求进行响应（针对 DELETE 请求）

针对2XX的状态码，一般设计到返回200即可，返回更加精细的状态码对于调用者来说意义不是很大。

针对301状态码（永久重定向）和302状态码（临时重定向，包括307），WEBAPI一般不会使用，一般由应用级别返回

关于303 See Other状态码表示参考另一个URL API。而不会重定向到URL。

304 Not Modified - HTTP缓存header生效的时候用，浏览器使用，WEBAPI不会使用。

4xx状态码表示客户端错误，主要有下面几种（WEB API 常用400,401,403,其他的状态码系统一般系统使用，WEB API一般不会主动设定，404除了API不存在以外，有些项目根据业务情况进行使用，例如查询在数据库中一定存在的数据，但是查询失败的场合下，会考虑抛出404）。

400 Bad Request：服务器不理解客户端的请求，未做任何处理（例如：请求Http的body出现错误，请求参数错误等）（WEB API可以选择使用）

401 Unauthorized：用户未提供身份验证凭据，或者没有通过身份验证。（WEB API认证机能经常使用）

403 Forbidden：用户通过了身份验证，但是不具有访问资源所需的权限。（权限管理相关使用。）

404 Not Found：所请求的资源不存在，或不可用。

410 Gone：所请求的资源已从这个地址转移，不再可用。

415 Unsupported Media Type：客户端要求的返回格式不支持。比如，API 只能返回 JSON 格式，但是客户端要求返回 XML 格式。

422 Unprocessable Entity ：客户端上传的附件无法处理，导致请求失败。

429 Too Many Requests：客户端的请求次数超过限额。

5xx 状态码（服务器发生异常时进行相关状态码抛出）

5xx状态码表示服务端错误。一般来说，API 不会向用户透露服务器的详细信息，所以只要两个状态码就够了。

500 Internal Server Error：客户端请求有效，服务器处理时发生了意外。

503 Service Unavailable：服务器无法处理请求，一般用于网站维护状态。

最好不要将业务的错误与状态码绑定，例如当某条数据为1时，不允许后续操作，则最好不要返回500错误。理由：实际上不是服务器发生错误，而是违反了某种业务规则。

可以考虑按照业务自定义状态码。

总结一下：在WebApi设计的时候，需要考虑与http的状态码保持一致。同时也不建议将所有业务异常与http状态进行捆绑。需要基于业务场景，需求精心设计

### 设计无状态的WebAPI

REST API 使用无状态请求模型。 HTTP 请求应是独立的并可按任意顺序发生，因此保留请求之间的瞬时状态信息并不可行。 信息的唯一存储位置就在资源内，并且每个请求应是原子操作。 此约束可让 Web 服务获得高度可伸缩性，因为无需保留客户端与特定服务器之间的关联。 任何服务器可以处理来自任何客户端的任何请求。 也就是说，其他因素可能会限制可伸缩性。例如，许多 Web 服务向后端数据存储写入数据，可能导致难以横向扩展。例如：用户的session信息，客户端上传的文件信息等。这种情况下采用第三方存储，例如缓存redis，共享存储等方式

### WebApi的设计粒度

一个最佳实践就是外粗内细的原则。对外的接口设计适当粗一点，一定程度保障是WebAPI的稳定，对于WebApi的内部实现，需要细一点（分解成多个方法）。

外粗的含义，例如业务需求，就有查询用户姓名，也有查询用户地址的需求。对于WebAPI来说不需要做成2个WebApi，一个查询用户姓名，一个查询地址。只需要做成查询用户信息（即把用户的相关非敏感信息一次性返回给客户端），由客户端根据不同的业务场景进行调用。好处API稳定，另外当有新的需求的时候，只需要再现有的WebAPI进行调整，而无需添加新的WebAPI。

还有一层考虑所有 Web 请求都会增加 Web 服务器的负载。 请求越多，负载就越大。 因此，请尽量避免使用公开大量小型资源的“琐碎”Web API。 此类 API 可能需要客户端应用程序发送多个请求才能找到它需要的所有数据。建议将数据非规范化，并将相关信息合并成可通过单个请求检索的较大资源。

内细的含义实际一个WebApi需要很多步骤，每个步骤定义成一个方法，每个方法按照单一职责原则，只完成一件事，便于为了复用和扩展。

### WebAPI的版本号

Web API 一直保持静态的可能性很小。 随着业务需求变化，可能会添加新的资源集合，资源之间的关系可能会更改，并且可能会修改资源中的数据结构。 虽然更新 Web API 以处理新的或不同的需求是一个相对简单的过程，但你必须考虑此类更改将对使用该 Web API 的客户端应用程序造成的影响。 问题在于尽管设计和实现 web API，开发人员可以完全控制该 API，开发人员不具有同等程度的控制可能由远程运营的第三方组织生成的客户端应用程序。 主要规则是要让现有客户端应用程序能够继续不变地正常运行，同时允许新客户端应用程序利用新功能和新资源。

在URL当中推荐加入API的版本号。例如：

<https://api.example.com/v1/>

## 性能设计（完善中）

### 定义

本文中提及性能设计主要是指针对开发层面（实现层）可能出现的性能问题，如何评估如何避免，并未从架构层面去考虑。

### 适用范围

|  |  |  |
| --- | --- | --- |
| 序号 | 项目种类 | 适用情况 |
| 1 | B/S系统 | **○** |
| 2 | C/S系统 | **○** |
| 3 | 移动端APP | **○** |
| 4 | 无界面系统（网关/批处理等） | **○** |
| 5 | 嵌入式系统 | **○** |
| 6 | 维护项目 | **○** |

### 术语表

性能指标：常见的衡量一个软件系统性能的指标有吞吐量、响应时间、资源使用率等。

吞吐量：指软件系统再单位时间内能处理的请求数量。

响应时间：用户感受软件系统为其提供服务所耗费的时间。对于网站系统来说，响应时间就是从点击了一个页面计时开始，到这个页面完全在浏览器里展现的这一段时间间隔，在这段响应时间内，软件系统在幕后经过了一系列的处理工作，贯穿了整个系统节点。根据“管辖区域”不同，响应时间可以细分为：服务器端响应时间、网络响应时间、网络响应时间。

资源使用率：指系统或网络资源的使用情况。常见的资源有：CPU占用率、内存使用率、磁盘I/O、网络I/O。

### 设计原则

忌过度设计原则：性能设计的目标是追求合适的性价比。

### 参考设计

以下项目中经常出现的性能问题，及相应的设计方案。

#### 前端

##### 页面初始化性能

* 减少HTTP请求次数
  + - * 强制删除对未使用资源的引用。
      * 推荐资源文件适度模块化，按需加载。
      * 推荐图片BASE64编码

将图片数据编码成 BASE64 的字符串，使用该字符串代替图像地址，使图片随着HTML的下载同时下载到本地，减少图片的请求次数。它适用的情况是浏览器连接服务器的时间 > 图片下载时间，也就是发起连接的代价要大于图片下载。

* 减少关键资源大小
  + - * 推荐图片格式转换

使用PNG8的格式，它可以在基本保持清晰度的情况下，减少图片的大小。

* + - * 推荐图片合并（CSS Sprites）

把多张图片整合到一张图片文件中，通过精确定位的方式使用图片。减少请求次数。

* + - * 推荐精简CSS和Javascript文件
* 推荐适当调整Javascript的位置

因为 HTML 的渲染和加载顺序是从上往下，JavaScript 放在尾部，这是传统的经验，它的好处是可以让页面优先渲染, 从而页面可以快速显示。

但是，这也可能导致这样一种情况：当页面已经渲染完毕时，网站的功能，无法使用。原因也很简单，就是 JavaScript 放在页面的尾部，还没来得及加载。

两种JavaScript放置方式，一种放在头部，一种放在尾部，一个牺牲了渲染速度，一个牺牲了用户体验。需要做权衡。

##### 数据渲染性能

* 推荐避免循环处理数据，使用数据源绑定，减少处理次数。

#### 后端

##### 文件读取

* 强制资源及时释放
* 推荐使用带缓冲的读写方式

##### 账票打印

* 推荐为了提高账票渲染速度，尽量避免大量数据的遍历处理，而是通过账票中间件提供的数据源绑定技术。

##### SQL性能

* 强制使用连接池，资源复用
* 强制使用变量绑定，减少硬解析
* 强制根据业务场景，合理创建索引
* 推荐编写SQL时，尽量避免使用会导致索引无效的写法

##### 批处理性能

* 推荐批处理通常都是用来处理大量数据的，需要制定一个合理的数据提交单位。如果全部等到事务或程序处理完毕一次性提交的话，会占用大量系统资源（如回滚段），拖慢系统速度。

## 国际化

在访问某些国际网站如Google时会自动显示中文页面，而有些国际网站则不会。这是因为Google实现了资源国际化，而有些网站则没有。

### 定义

如果一个系统会被多个国家与地区使用，则需要解决这些国家与地区间的文化差异的问题。文化差异不仅仅是语言的差异，还包括生活习惯等差异。例如中国与美国将小数“零点一”记作“0.1”，而欧洲习惯记作“0,1”。这些问题的解决便是资源国际化。显然，这些问题不是简单做几套不同语言的软件系统就能解决的，它是个很复杂的问题。资源国际化有两个常用的术语I18N与L10N。

I18N即资源国际化，全称为Internationalization，因为首字母I与末字母N之间共18个字母，又称为I18N。通俗地讲，资源国际化就是要让这个软件产品使用国际环境，如语言、文化、使用习惯等。

L10N为资源本地化，全称为Localization，因为首字母L与末字母N之间共10个字母，又称为L10N。资源本地化就是要让这个软件产品使用当地的环境，如语言、文化、使用习惯等。

资源国际化与资源本地化这两个概念并没有太大的差别。如果一套系统最初只有英语界面，那么为了适应中国地区用户而使系统支持中文的过程便可视为本地化。

### 适用范围

|  |  |  |
| --- | --- | --- |
| 序号 | 项目种类 | 适用情况 |
| 1 | B/S系统 | **○** |
| 2 | C/S系统 | **○** |
| 3 | 移动端APP | **○** |
| 4 | 无界面系统（网关/批处理等） | **✕** |
| 5 | 嵌入式系统 | **✕** |
| 6 | 维护项目 | **✕** |
|  |  |  |

### 国际化的特征

一个经过国际化的软件具备如下特征：

* 可以快速的本地化；
* 借助本地化的数据，相同的程序可以运行在世界各地；
* 诸如状态提示、界面中的标题等文本元素，并非硬编码到软件中，而是存储到源代码之外，且能动态改变；
* 不需要重新编译软件就能添加新的语言；
* 货币、日期等信息的表现形式，适应于用户所在的地区和语言；

### 需要本地化的元素

除了以上提到的货币和日期，还有很多元素与文化、地域、语言相关，比如：书写方向、声音、颜色、图形、图标、时间、数字、度量、电话号码、敬语、头衔、邮政地址、分页、排序方法、输入处理、方言、法规、道德和习惯等。

### 国际化参考设计

#### 设计思想

* 创建资源包和资源文件
* 设置资源文件
* 选择和加载语种
* 设置语言环境

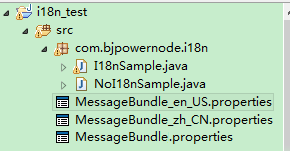
#### B/S模式参考设计

##### 创建资源文件

以JAVA为例，Java 将不同语言的文本存储在后缀为 .properties 的文件中，其格式为：<资源名>\_<语言代码>\_<国家/地区编码>.properties

其中，语言编码和国家/地区编码都是可选的；以 <资源名>.properties 命名的国际化资源文件是默认的资源文件。

例如：



##### 设置资源文件

* 国际化的英文环境的properties文件MessgesBundle\_en\_US.properties：

username=username

password=password

submit=submit

* 国际化的中文环境的properties文件MessgesBundle\_zh\_CN.properties：

username=\u7528\u6237\u540D

password=\u5BC6\u7801

submit=\u63D0\u4EA4

注意：要用Unicode文字。

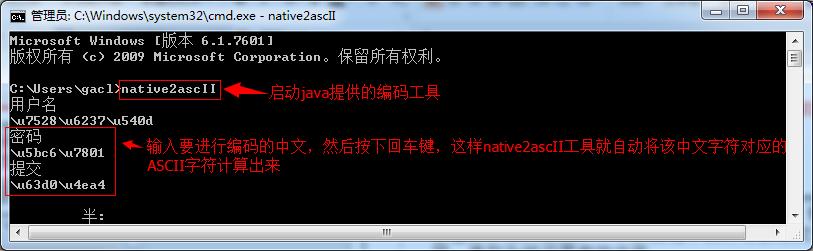
* 国际化的默认的properties文件MessgesBundle.properties：

username=Username

password=Password

submit=Submit

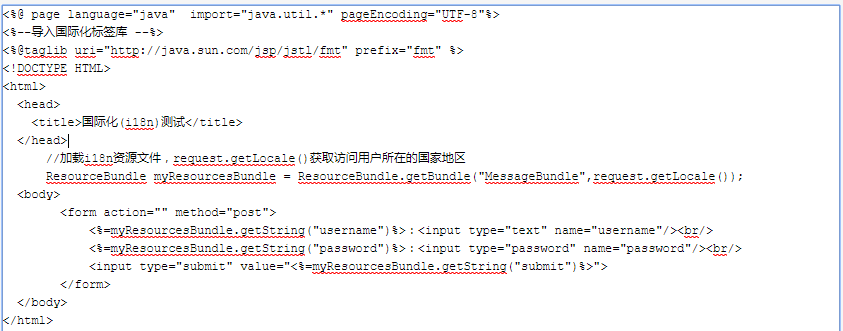
java提供了一个native2ascII工具用于将中文字符进行编码处理，native2ascII的用法如下所示：



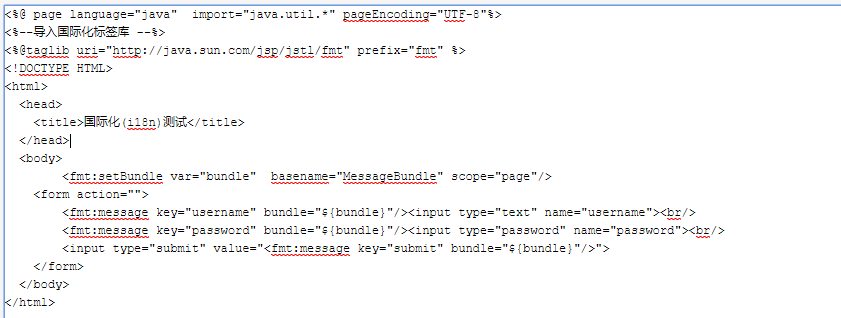
##### 选择和加载语种

在 Java 中，后台用 java.util.Locale 对象选择区域设置，获取的是系统区域的设置，一般在C/S模式下使用；而B/S模式通常在前台用request.getLocale获取浏览器的语言设置；而 java.util.ResourceBoundle 则用于加载本地化资源文件。

如下的代码例子是通过两种方式实现文本的国际化，一个是普通的标签：

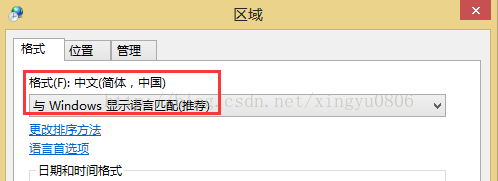


另一种是通过国际化标签库实现固定文本的国际化：

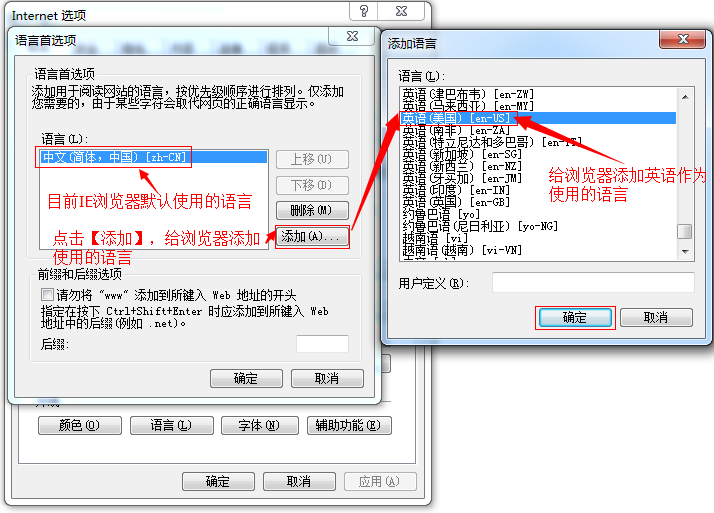


##### 设置语言环境

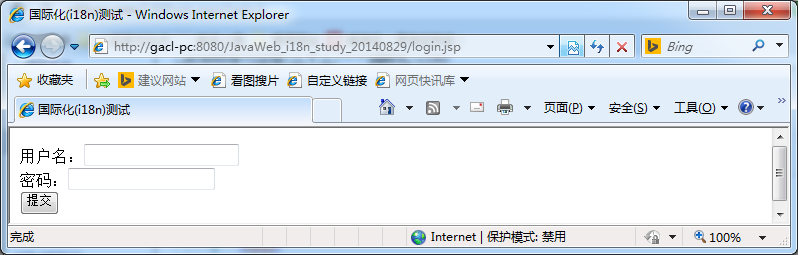
C/S模式时，当本机是中文的时候，去读取中文的国际化资源文件，对应显示的是中文，设置如下：



B/S模式时，通过IE浏览器-工具-Internet选项中，设置语言首选项为中文，去读取中文的国际化资源文件：



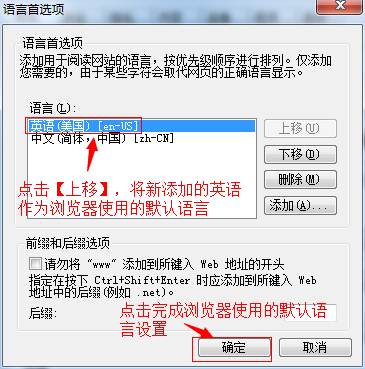
上面例子的运行结果：



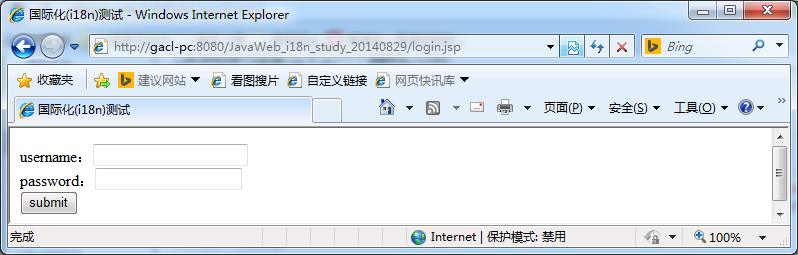
C/S模式时，当本机调成英文的时候，去读取英文的国际化资源文件，对应显示的是英文，设置如下：



B/S模式时，通过IE浏览器-工具-Internet选项中，设置语言首选项为英语，去读取英文的国际化资源文件：



上面例子的运行结果：

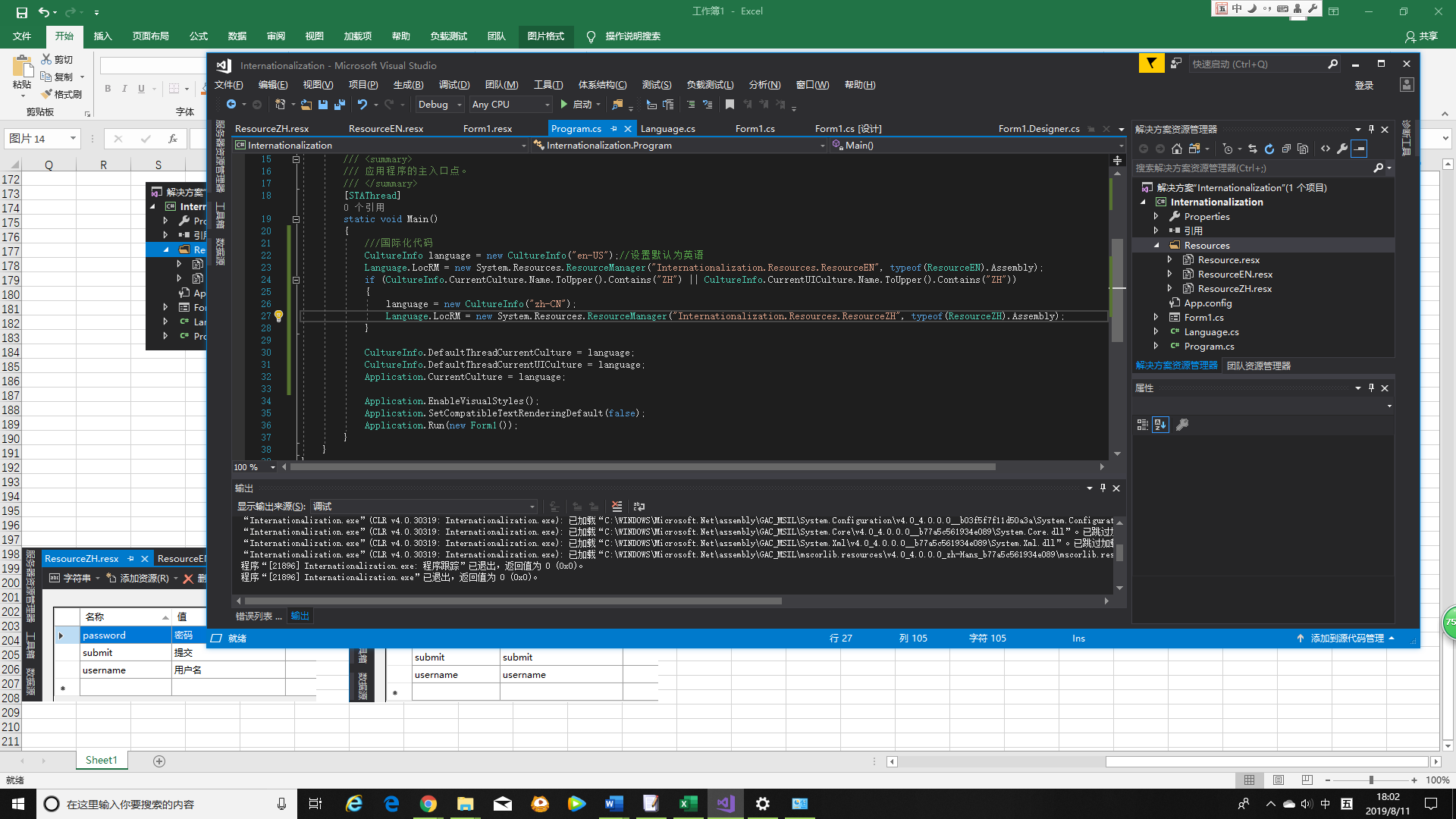


如果没有找到上述设置的中文或英文语言资源包，则显示默认资源文件的内容。

#### C/S模式参考设计

##### 创建资源文件

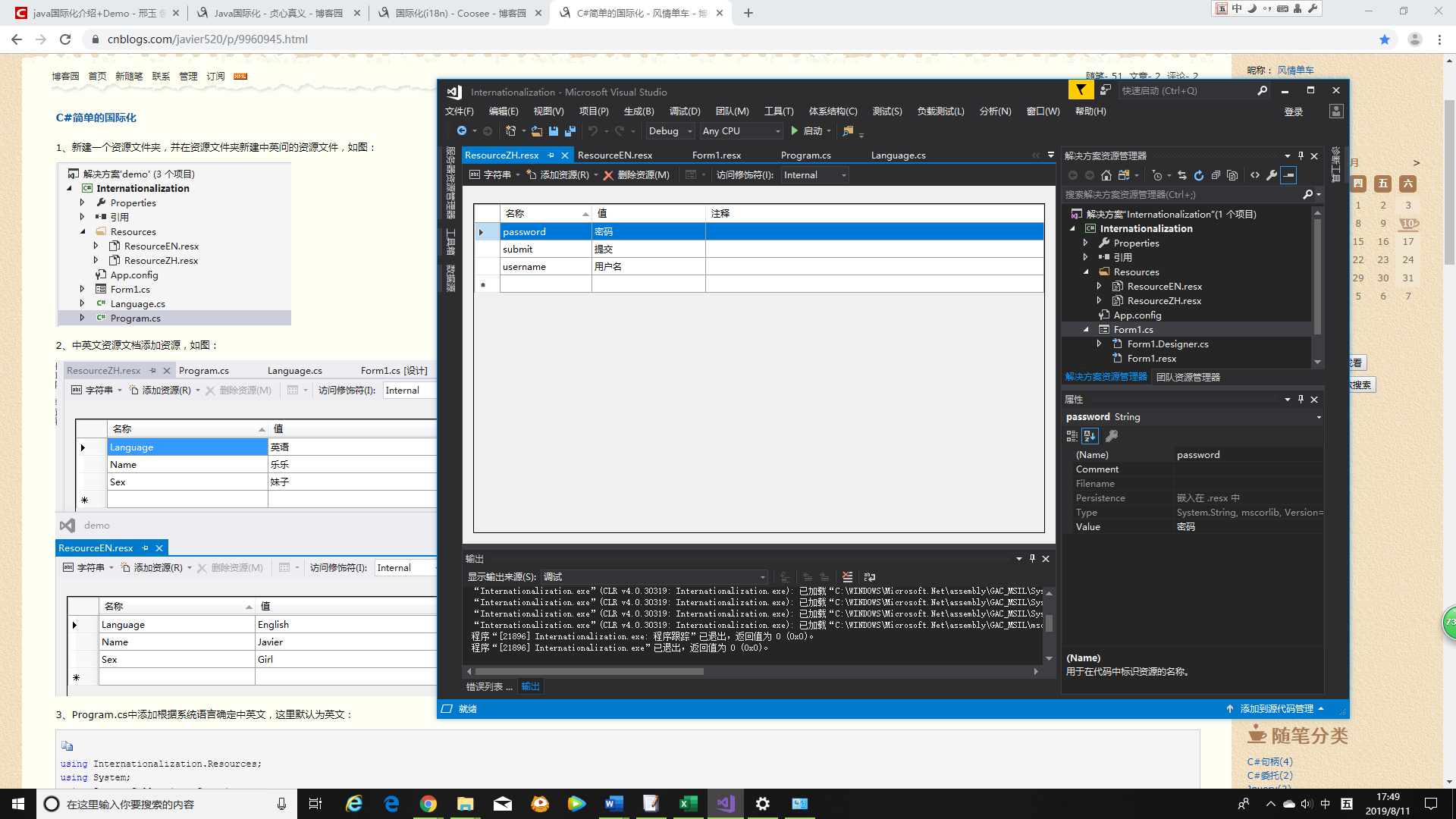
以C#为例，C#将不同语言的文本存储在后缀为 .resx 的文件中。手动向项目中添加resource文件，部署目录参考如下：



另一种创建方式是通过Visual Studio自动生成对应的语言的resource文件，这样每一个窗体的资源文件都部署在自己的窗体下，考虑到以后的扩展和开发时的效率，一般不采用此方式。

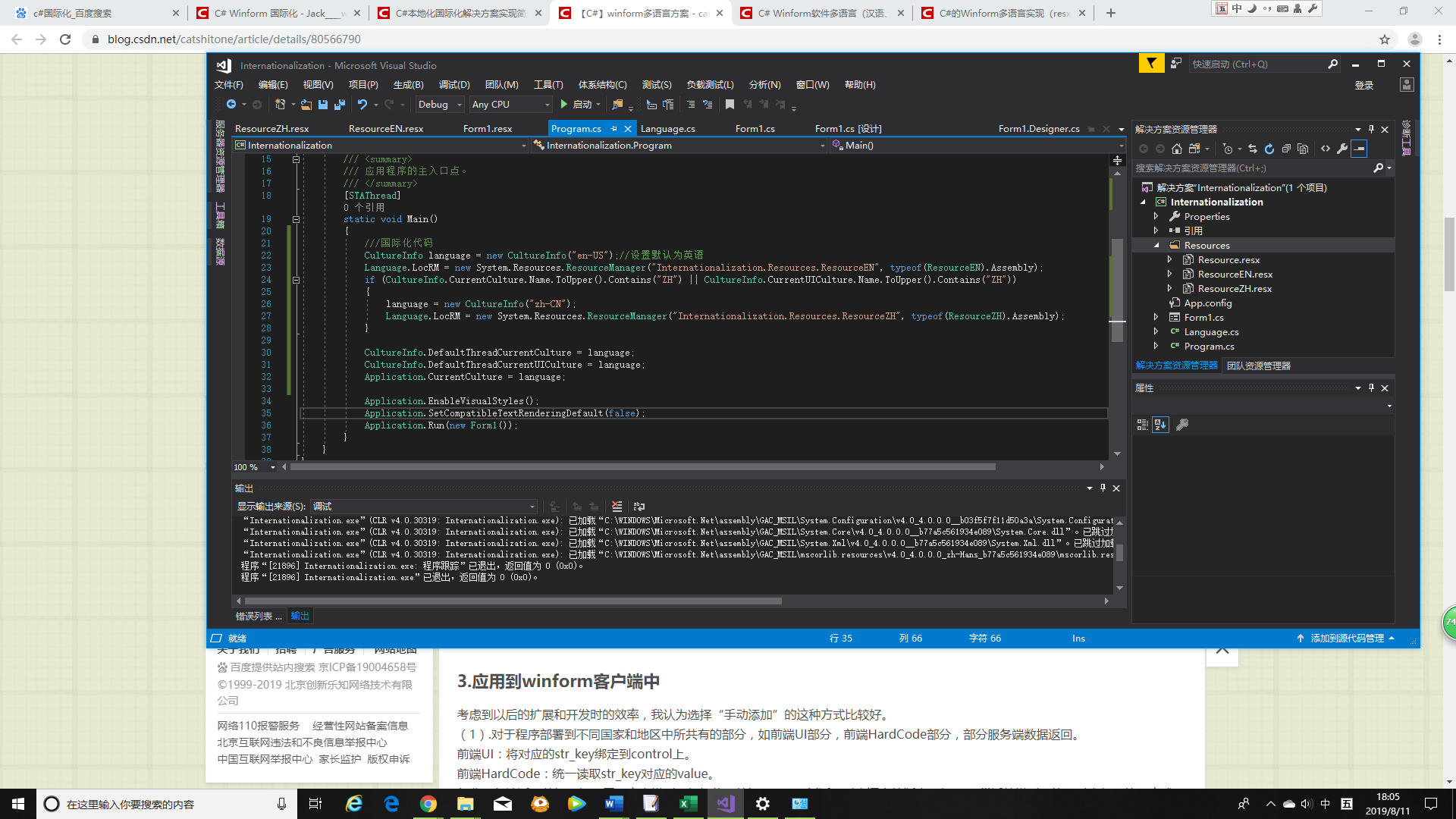
##### 设置资源文件

为资源文档添加资源：



##### 选择和加载语种

使用时，通过CultureInfo获取当前系统设置的区域性信息和当前系统的显示语言，通过ResourceManager来读取资源文件：



##### 设置语言环境

C/S模式的设置语言环境的方式是在系统区域里设置语言，就会显示不同的语言环境的结果。

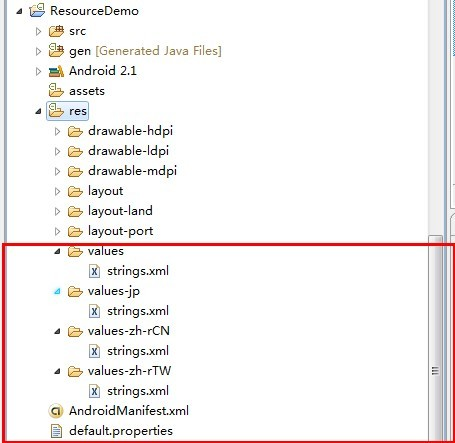
#### 移动端APP参考设计

##### 创建资源文件

Android系统的国际化需要在应用所在目录res文件夹下建立针对不同语言或地区的values文件夹，格式为：values-语言代码-r国家代码，如values-zh-rCN简体汉语，values-zh-rTW繁体，values-jp日语等。(表示中文和中国的配置选项是 zh-rCN; 表示英文和美国的配置选项是en-rUS. 表示zh 和 en 表示中文和英文；CN 和 US 表示中国和美国; 前面的r 是必须的)。

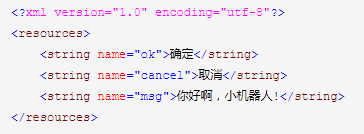
如果需要让图片也实现国际化，还要为drawable目录添加几个不同语言国家的版本，不同drawable文件夹的命名方式为：drawable-语言代码-r国家代码。

目录结构参考如下图所示:



##### 设置资源文件

在values-zh-rCN文件夹下建一个strings.xml代表中文资源：



在valuesen-rUS文件夹下也同样建一个strings.xml代表英文资源。

##### 选择和加载语种

在布局文件中使用这些字符串资源：



接着在activity中通过setContentView(R.layout.main)加载布局文件。

##### 设置语言环境

通过android模拟器的语言设置来显示中文或英文效果。

#### 国际化的切换方式

无论是B/S模式，还是C/S模式，国际化的切换，既可以根据系统环境进行自动切换，也可以在画面上做出可以选择语言的切换控件，修改语言变量，从而读取不同的配置文件，不同的资源文件来达到切换语言的目的。

#### 国际化设计的注意点

* 问号，方块和乱码的问题在国际化测试中很常见，显示方框是因为程序没有合适的字体。问号是从Unicode到ANSI的转换不对。问号？的产生是因为Unicode到代码页转换失败。乱码是CodePage对应的不对，1，系统或者软件缺少对应的字符编码，需要安装对应的字符编码表。2.二进制码进入计算机后被各种不同编码解释为不同符号。所以打开文件必须先知道编码方式。如果选择错误的代码页，就会显示乱码。
* Android的屏幕适配，Android设备屏幕分辨率很多，为了适应不同终端设备的不同分辨率，从而使布局显示更加合理和准确，需要在res目录下建立对应不同设备分辨率的文件夹。文件夹格式：layout-长x宽，数字大的放在前面。
* B/S模式，语言切换时，画面布局和样式显示不正常，可以把css作为网页布局文件一般嵌入到html页面或者jsp页面中，它包括图片的国际化和布局样式的国际化。例如创建style.css和style\_en\_US.css等。
* C/S模式，语言切换时，画面布局不正常，AutoSize 属性导致控件按其内容调整自身的大小。TableLayoutPanel 控件提供了按比例调整大小的能力，它使控件可以在调整大小以适应其内容时保持相对比例不变。

#### 国际化其它元素的参考设计

每一种开发语言都提供了对应的格式化类，以Java为例：

##### 日期

DateFormat 类可以将一个日期/时间对象格式化为表示某个国家地区的日期/时间字符串。

DateFormat 类除了可按国家地区格式化输出日期外，它还定义了一些用于描述日期/时间的显示模式的 int 型的常量，包括FULL, LONG, MEDIUM, DEFAULT, SHORT，实例化DateFormat对象时，可以使用这些常量，控制日期/时间的显示长度。

实例化DateFormat类有九种方式，以下三种为带参形式，下面列出的三种方式也可以分别不带参，或只带显示样式的参数。

* getDateInstance(int style, Locale aLocale)：以指定的日期显示模式和本地信息来获得DateFormat实例对象，该实例对象不处理时间值部分。
* getTimeInstance(int style, Locale aLocale)：以指定的时间显示模式和本地信息来获得DateFormat实例对象，该实例对象不处理日期值部分。
* getDateTimeInstance(int dateStyle, int timeStyle, Locale aLocale)：以单独指定的日期显示模式、时间显示模式和本地信息来获得DateFormat实例对象。

##### 数字格式化

NumberFormat类可以将一个数值格式化为符合某个国家地区习惯的数值字符串，也可以将符合某个国家地区习惯的数值字符串解析为对应的数值。

* getNumberInstance(Locale locale)：以参数locale对象所标识的本地信息来获得具有多种用途的NumberFormat实例对象。
* getIntegerInstance(Locale locale)：以参数locale对象所标识的本地信息来获得处理整数的NumberFormat实例对象。
* getCurrencyInstance(Locale locale)：以参数locale对象所标识的本地信息来获得处理货币的NumberFormat实例对象。
* getPercentInstance(Locale locale)：以参数locale对象所标识的本地信息来获得处理百分比数值的NumberFormat实例对象。

##### 设计思路

日期、时间、货币、数字等的表示，可以结合上述B/S模式、C/S模式等的设计参考，根据获取系统的语言进行判断，根据判断结果，在不同分支，去表示对应语言的日期、时间、货币、数字等。

## Session管理设计

# 实现设计(Design Definition)-公用模块域

## 弹出消息(Messages)

### 定义

弹出消息是指应用程序发出的，需要用户马上知晓或者作出选择的信息。按照Windows的界面设计语言，弹出消息包括四种：错误消息(errors)、警告消息(warning)、确认消息(confirmations)、提示消息(notifications)。

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 模态 | 典型按钮 | 备注 |
| 错误消息(errors) | **○** | 【确定】 |  |
| 警告消息(warning) | **○** | 【确定】【是、否】【确定、取消】 |  |
| 确认消息(confirmations) | **○** | 【是、否、取消】【是、否】 |  |
| 提示消息(notifications) | **✕** | 无按钮 | 不中断用户操作，允许用户忽略的信息。 |

### 适用范围

|  |  |  |
| --- | --- | --- |
| 序号 | 项目种类 | 适用情况 |
| 1 | B/S系统 | **○** |
| 2 | C/S系统 | **○** |
| 3 | 移动端APP | **△** |
| 4 | 无界面系统（网关/批处理等） | **✕** |
| 5 | 嵌入式系统 | **✕** |
| 6 | 维护项目 | **✕** |
|  |  |  |

### 术语表

消息框样式：包括消息框的视觉外观、按钮及其作用定义。

### 区别与联系

弹出消息（Messages）和对话框（dialog boxes）不同，弹出消息是一种以“通知”为主要目的的界面元素固定的弹出窗口，对话框通常含有某种业务处理，具有相对复杂的交互元素。

在设计消息处理方案时，需要考虑与系统的异常处理方案协调一致。

### 弹出消息的设计原则

1. 统一性原则：所有代码都只能通过调用公共方法显示弹出消息，通过公共方法保证弹出消息的视觉特征、交互行为的一致性。
2. 可管理原则：系统中的弹出消息文本和消息类型，必须使用公共类集中定义。
3. 易诊断原则：错误消息要能够有利于问题的查找与分析，要能够通过错误消息识别故障原因。

### 弹出消息参考设计

通常，一个系统的消息处理设计需要考虑以下几个方面：

* 定义弹出消息的场合
* 消息代码规则
* 消息内容语言规范
* 消息内容的定义方式与生成方式
* 实现弹出消息的公共方法
* 消息框元素构成与视觉设计

#### 弹出消息的场合

在进行设计时，需要结合当前系统的需求和特点，定义程序需要主动弹出消息的情况，用来指导开发人员，同时保证系统行为一致，易于用户理解和使用。下述内容为一般情况的设计和定义，供参考。

* 显示用户操作的结果（成功/失败）。当系统执行了某种对未来产生影响的操作以后，告知用户操作的结果。比如：保存成功、发送成功等。在一个具体的系统中，应该定义统一的原则，指导开发人员，最终使得系统对用户呈现出一致的行为表现。
* 对于关键操作进行确认，防止误操作。当用户选择执行某种关键操作，特别是不可撤销的操作时，应该使用弹出消息告知用户该操作的风险，让用户确认是否要执行该操作。比如：删除数据前，通过弹出消息告知用户“数据一旦删除不可恢复，是否继续？”
* 显示系统错误。当系统运行过程中发生错误时，要及时通过弹出消息通知用户，告知用户系统发生错误的原因，并给出应对办法。
* 与用户当前操作无关的消息。系统内部产生的某种状态，需要通知用户，但是该消息与用户正在进行的处理无关，所以采用不打断用户操作的方式显示。比如：“您有一条新的待办任务”
* 其他情形。根据系统不同，还需要通过弹出消息告知用户的其他情形，比如：有的系统当用户登陆以后，通过弹出消息告诉用户“密码即将过期，请在10天内修改密码”。

上述各种显示弹出消息的场合，与消息提示框的种类的关系如下表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 提示场合 | 错误 | 警告 | 确认 | 提示 |
| 操作结果 | ○ |  | ○ |  |
| 确认 |  | ○ | ○ |  |
| 系统错误 | ○ |  |  |  |
| 与当前操作无关的消息 |  |  |  | ○ |

#### 消息代码规则

消息代码是指为系统中的全部消息编制统一规则的代码，代码和消息一一对应。消息代码除了在程序中作为常量名称指代消息文本以外，开发人员还可以根据消息代码来推测系统行为进而确定系统的故障点，同时还是一种快速交流信息的手段，能够准确地传达系统弹出了一条什么消息。因此，在错误消息中显示消息代码是有必要的，消息代码的编码规则也要有利于诊断。

**推荐** 以下的消息代码设计可供在项目中参考，本设计着重考虑了系统错误的诊断需要。

|  |  |  |
| --- | --- | --- |
| 模块(机能)代码  （2位字母） | 类型代码  （1位字母或数字） | 流水号  （3位数字） |
| 区分系统中不同的模块，全系统的公共消息用CM（英文commomn） | 区分消息类型：  W：警告  C：确认  T：提示  0:运行时异常  1:数据库异常  2:文件处理异常  8:外围系统异常  9:业务异常  3～7:其他  （数字代表错误消息，与异常处理的设计关联，字母代表一般消息） | 000～999顺序定义 |

在定义消息代码时，以下一些字母容易混淆不建议使用：

* 字母I，容易和1混淆
* 字母O，容易和0混淆
* 字母L，容易和1混淆
* 字母Q，容易和数字0字母O混淆

#### 消息内容语言规范

弹出消息中的内容文本，是人机界面的重要组成部分，良好的消息语言能够让用户觉得系统更可靠、更友好、更智能，不好的消息语言会引起用户产生误解、觉得系统难以使用、甚至对系统产生排斥心理。一般来说，应遵循以下一些原则：

* 文体统一原则：业务系统一般使用书面语，日语的情况下要注意简体、敬体不能混用，一般统一成敬体。
* 可理解原则：消息内容要使用最终用户能够理解的语言，术语要统一（参考项目业务术语表）。错误消息的信息量要适度，不使用引起用户误解或者无法理解的计算机专业词汇。比如：“数据错误”是用户能够理解的语言，但是“主键冲突”是用户无法理解的语言。对于异常情况，一定要明确告知用户操作结果（比如“数据更新失败”），然后才是失败的原因。
* 诊断性原则：当系统发生故障时，应该明确告知用户错误产生的主要原因，这样有助于系统的运维人员定位故障，从而尽快排除故障。诊断性原则要与可理解原则平衡，输出适当的信息量，同时可以采用“错误代码”作为重要的补充信息，让运维/开发人员能够快速诊断问题的原因。
* 充分性原则：消息内容要能够全面清晰地传达信息。警告信息应该包括对于操作后果的陈述，比如：“数据一旦删除不可恢复，是否继续？”。错误信息应该包括操作结果、原因、处理办法，比如：“数据更新失败。当前数据已经被其他用户更新，请确认后重新操作。”
* 帮助性原则：消息内容要能够引导用户做出正确的操作。比如：对于临时性的系统错误，应该提示用户“请稍后重试”；对于较为严重的错误，应该提示用户“请联系管理员”。

#### 消息内容的定义方式与生成方式

##### 消息内容定义

**强制** 为了提高系统中产生的弹出消息的可维护性和可管理性，全部的消息文本要求集中定义。

下表对比了几种定义方式的优缺点：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 定义方式 | 优点 | 缺点 | 备注 |
| 1 | 在代码中用常量定义 | 编码时可以借助IDE功能列出ID，比较方便，ID敲错了能够通过编译器检查发现。 | 修改消息定义需要重新编译、发布程序。 | **推荐** |
| 2 | 在数据库中定义 | 可以不重启程序，动态修改消息定义。  易于国际化。 | 编码时不方便，ID有错误难以检查。  需要在建库脚本中维护消息定义。 |  |
| 3 | 在配置文件中定义 | 修改消息定义不用重新编译、发布程序。  易于国际化。 | 编码时不方便，ID有错误难以检查。 |  |
| 5 | 混合定义，在数据库或者文件中定义内容，在代码中定义常量关联ID。 | 修改消息定义不用重新编译、发布程序。  编码时方便引用。 | 需要同时维护常量类和消息定义，导致代码逻辑分散。 |  |

**推荐** 定义一个专用类，用来保存：弹出消息文本内容、弹出框样式。然后用常量的方式，集中定义消息文本内容和对应的弹出框样式。

##### 消息内容生成

有一些情况下，消息内容需要通过消息文本和业务数据合成，比如：“当前共选择了()条数据，是否批量删除？”。在显示这条弹出消息时，需要填入当前操作中涉及到的数据数量。

只有在必须引入业务数据的情形下，才使用消息文本和成。

**禁止** 不允许定义特别灵活的弹出消息，然后通过合成消息生成大量不同的消息内容，这将会导致消息内容定义的分散，导致不可管理。比如定义了一条消息文本“()操作成功”，然后合成出“保存操作成功”、“删除操作成功”等等消息。

#### 实现弹出消息的公共方法

**强制** 为了保证系统行为的一致性，避免开发人员自行编码引起的诸多问题，提高可维护性，要求系统中必须实现统一的弹出消息公共方法，所有弹出消息必须经由统一的公共方法实现。

**推荐** 弹出消息的公共方法参数设计参考下表：

|  |  |  |
| --- | --- | --- |
| 序号 | 参数 | 说明 |
| 1 | 消息ID | 代表提示消息的常量 |
| 2 | 消息合成参数 | 支持多个 |

开发弹出消息的公共方法时，推荐采用下述原则：

1. 消息ID和弹出消息框的样式严格绑定，在定义消息文本内容的同时指定（参见前文消息内容定义），不再另外设置参数控制弹出框的样式。这样做的好处是集中控制，减少对开发人员编码时的依赖，避免犯错。
2. 对消息合成参数与消息文本的匹配性进行检查，如果发现个数不相符，需要抛出业务异常。

#### 消息框样式

**强制** 要求在实现设计阶段，结合系统业务需求，对本系统中允许使用的弹出框的样式进行穷举定义。需要定义的内容参考下表：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 序号 | 弹出消息框名称 | 按钮 | 视觉定义 | 对话框标题 | 右上角X按钮 |
| 1 | 警告1 | 确定 |  | 警告 | 无 |
| 2 | 警告2 | 是、否 |  | 警告 | 无 |
| 3 | 警告3 | 是、否、取消 |  | 警告 | 有 |
| 4 | 确认1 | 是、否 |  | 确认 | 无 |
| 5 | 确认2 | 是、否、取消 |  | 确认 | 有 |
| 6 | 错误 | 确定 |  | 错误 | 无 |

**推荐** 系统中的全部消息框样式，可以使用枚举(enum)类型进行定义。

#### 参考资料

* 微软设计语言关于弹出消息的使用指南：

<https://docs.microsoft.com/en-us/windows/win32/uxguide/messages>

## 数据验证

### 数据验证的定义和分类。

#### 术语表

输入验证 ： 指对程序入口数据进行验证，包括用户输入的数据或者API调用传递的参数等。

逻辑验证 ： 指对程序中数据的业务逻辑进行验证。

DB验证：指数据验证过程需要提取数据库中的数据进行比较。

非DB验证：指数据验证过程不需要提取数据库中的数据。

前端验证：指发生数据验证的位置在程序设计的前端，比如使用js脚本在HTML页面验证数据。

后端验证：指发生数据验证的位置在程序设计的后端，比如HTTP开发中，使用java验证web接口的参数。

单项目验证：指验证的过程中只对一个数据进行某一种类型的验证。

多项目验证：指验证的过程中需要两个以上的数据才能进行的验证。多个数据的来源可以是数据库的数据，也可以不是数据库的数据。

#### 定义和分类

数据验证是为了保证数据的正确性和完整性进行的程序操作。

从完成的功能上分类：输入验证和逻辑验证。

从程序架构中完成验证的功能位置上分类：前端验证和后端验证

#### 适用范围

|  |  |  |
| --- | --- | --- |
| 序号 | 项目种类 | 适用情况 |
| 1 | B/S系统 | **○** |
| 2 | C/S系统 | **○** |
| 3 | 移动端APP | **○** |
| 4 | 无界面系统（网关/批处理等） | **△** |
| 5 | 嵌入式系统 | **△** |
| 6 | 维护项目 | **✕** |
|  |  |  |

#### 数据验证的设计原则

数据验证是不能程序不能缺少的部分。完全信赖输入的数据和完全依靠数据库约束去校验数据都是错误的方式。主要原则如下：

1. 数据验证的基本准则是本着不相信用户任何输入
2. 先进行输入验证，再进行逻辑验证
3. 输入验证先进行单项目验证，再进行组合项目验证
4. 逻辑验证先进行和数据库不相关的验证，再进行和数据相关的校验
5. 先进行前端验证，再进行后端验证
6. 即使在输入验证上前后端做了相同的事情，后端验证也不能缺失。
7. 程序验证即使和数据库约束验证相同，也不能缺失程序验证。
8. 数据验证设计时要和Message消息功能结合，得以给用户返回验证失败信息。

### 输入验证和逻辑验证

#### 输入验证

输入验证对程序健壮性、容错性和安全性起着至关决定的作用。所以输入验证数据的范围是一个程序、一个功能甚至一个函数的所有入口数据。所以输入验证也必须对非输入项目进行校验。输入验证的适用范围如下：

|  |  |  |
| --- | --- | --- |
| 序号 | 项目种类 | 适用情况 |
| 1 | 用户输入项目 | **○** |
| 2 | 函数参数 | **○** |
| 3 | HTTP请求的parameter（B/S） | **○** |
| 4 | 其他TCP或者SOAP协议请求参数 | **△** |
| 5 | Windows窗口程序跳转参数（C/S） | **△** |
|  |  |  |

输入验证的种类如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 验证种类 | 示例 | 类型 |
| 1 | 必须输入 |  | **单项目** |
| 2 | 指定类型 | 整数、浮点、Boolean | **单项目** |
| 3 | 值域范围 | 整数、浮点 | **单项目** |
| 4 | 指定数据格式 | 浮点数的整数和小数长度、时间格式 | **单项目** |
| 5 | 字符、字节长度 |  | **单项目** |
| 6 | 特定格式（电话号码、邮箱、URL） |  | **单项目** |
| 7 | 正则表达式其他格式 | 信用卡号、流水单号 | **单项目** |
| 8 | 大小比较 | 开始时间和结束时间比较 | 多项目 |
| 9 | 类型下值域范围 | 成绩A的情况下，分数范围为100-90分；成绩B的情况下，分数范围为89-70分 | 多项目 |
|  |  |  |  |

一般推荐的输入验证的流程如下 ：

指定类型

值域范围

长度范围

多项目验证

特定格式

正则验证

指定类型

必须输入

#### 逻辑验证

逻辑验证是对程序业务逻辑的实现。逻辑验证多数是需要操作数据库实现的，部分逻辑验证是会和数据库的约束造成重复。但是考虑到数据库的操作压力，不建议使用数据库异常方式进行逻辑验证。而是建议使用程序进行逻辑验证。如果在一个业务中存在多个逻辑验证，考虑到程序的可读性和执行性能，不建议使用过多子查询一次性读取很多的数据，然后进行逻辑验证。建议一次查询一张表中的数据，进行验证后，再继续下一个验证。

输入验证

查询表B

逻辑验证B

逻辑验证A

查询表A

对一个完整功能来说，建议验证的顺序是：先进行输入验证，通过后再进行逻辑验证

逻辑验证

### 数据验证的共通实现

#### 数据验证实现的方案

数据验证实现的方案主要有三种

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 验证种类 | 特点 | 示例 |
| 1 | 代码实现 | 不使用任何框架，灵活。但是代码量多 |  |
| 2 | 注解(装饰器)实现 | 使用AOP+注解实现，代码量少。但是灵活性不足。 | Java的hibernate validate |
| 3 | 正则表达式 | 利用正则表达式的强大，代码量少。但是完成复杂正则书写难度高 | Javascipt前端验证 |

考虑程序的阅读性，推荐使用注解(装饰器)实现和正则表达式实现。

常用正则表达式的整理（JS版本）

|  |  |
| --- | --- |
| 验证内容 | 正则表达式 |
| 整数或者小数 | ^[0-9]+\.{0,1}[0-9]{0,2}$ |
| 只能输入数字 | ^[0-9]\*$ |
| 只能输入m~n位的数字 | ^\d{m,n}$ |
| 只能输入零和非零开头的数字 | ^(0|[1-9][0-9]\*)$ |
| 只能输入由26个英文字母组成的字符串 | ^[A-Za-z]+$ |
| 只能输入汉字 | ^[\u4e00-\u9fa5]{0,}$ |
| 验证Email地址 | ^\w+([-+.]\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*$ |
| 验证URL | ^http://([\w-]+\.)+[\w-]+(/[\w-./?%&=]\*)?$ |
| 验证电话号码 | ^(\(\d{3,4}-)|\d{3.4}-)?\d{7,8}$ |
| 验证手机号 | ^1[34578]\d{9}$ |
| 验证身份证号 | ^\d{18}$ |
| 验证一年的12个月 | ^(0?[1-9]|1[0-2])$ |
| 验证一个月的31天 | ^((0?[1-9])|((1|2)[0-9])|30|31)$ |
| 验证强密码，最少6位，包括至少1个大写字母，1个小写字母，1个数字，1个特殊字符 | ^.\*(?=.{6,})(?=.\*\d)(?=.\*[A-Z])(?=.\*[a-z])(?=.\*[!@#$%^&\*? ]).\*$ |
| 验证IPv4地址 | ^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$ |
| 验证日期 | ^\d{4}(\-)\d{1,2}\1\d{1,2}$ |
| 验证车牌号 | ^[京津沪渝冀豫云辽黑湘皖鲁新苏浙赣鄂桂甘晋蒙陕吉闽贵粤青藏川宁琼使领A-Z]{1}[A-Z]{1}[A-Z0-9]{4}[A-Z0-9挂学警港澳]{1}$ |
|  |  |

#### 数据验证和message消息的结合

数据验证过程中，如果验证不通过的情况下，输出结果有两种 ：

1.通知用户显示错误消息。

所以设计实现时需要和【4.1弹出消息（Messages）】的内容相结合，如果check不通过的情况下，返回相应的message消息。用于进行消息在页面显示。

2.不能进行显示部分要打入异常日志。

所以设计时需要和【4.4 日志处理】的内容相结合。例如，用户点击修改按钮，传递数据id到查询功能，此时对id进行非空数据验证，如果不通过的情况下，要将错误消息写入日志中。

#### 前端验证和后端验证的设计原则

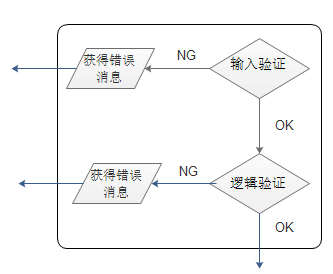
前端验证和后端验证是按照验证程序逻辑在程序中位置进行区分的。从功能上讲前端验证主要是完成和用户的交互，用于给用户显示错误信息。后端验证主要保证的是程序的功能性、健壮性和安全性。所以前端验证和后端验证的主要设计原则如下 ：

不管有没有前端验证，后端验证必须具有输入验证和逻辑验证。

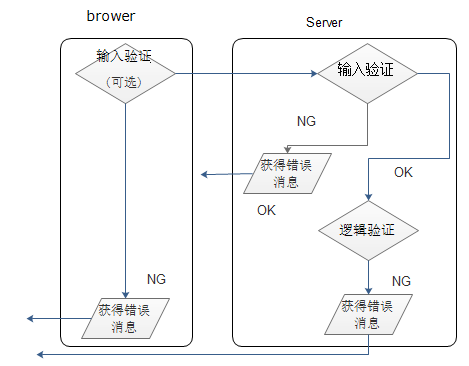
#### 各种系统架构下数据验证的实现

1. C/S

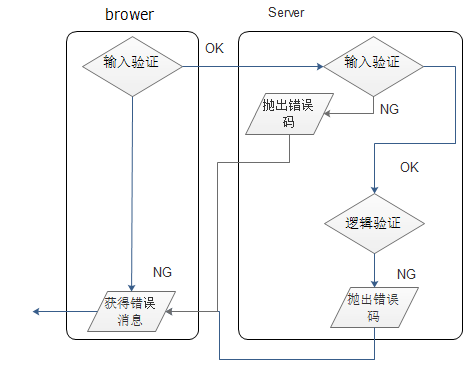
client



1. B/S - 传统MVC模式



1. B/S – 前后端分离模式



## 异常处理

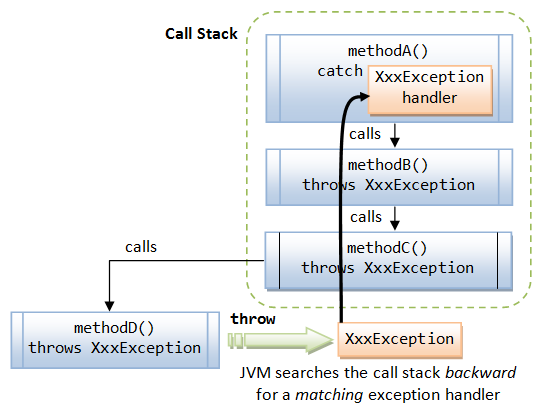
### 定义

在本文中，异常处理泛指在程序遇到错误后跳出原运行路径的分支结构，当异常发生时，程序执行的流程发生改变，程序控制权从正常业务处理转移到异常处理逻辑。

对于Java、.Net等高级语言，提供了try语法和throw语法来实现异常抛出机制。对于C语言，没有专门进行错误处理的语法结构，但是提供了若干函数来帮助程序员对错误进行截获和处理。

编程语言的异常机制，其本质是当运行过程中发生错误时，提供一种跳出原有执行序列的机制，从而把程序的正常流程代码，和错误(例外)处理代码分开，最终达到提高代码的可读性和可维护性的目的。

下图说明了通过异常抛出机制，实现了跨越调用层级，直接穿透到最上层的效果，也就是可以在最上层集中处理异常情况，从而分离了正常逻辑和错误处理逻辑的代码，提高了可读性、可维护性、内聚性。



### 适用范围

|  |  |  |
| --- | --- | --- |
| 序号 | 项目种类 | 适用情况 |
| 1 | B/S系统 | **○** |
| 2 | C/S系统 | **○** |
| 3 | 移动端APP | **○** |
| 4 | 无界面系统（网关/批处理等） | **○** |
| 5 | 嵌入式系统 | **○** |
| 6 | 维护项目 | **○** |
|  |  |  |

### 术语表

### 区别与联系

异常处理方案与日志处理方案、提示消息方案有着密切的关系，需要协同、统筹设计。通常在捕获异常以后，要依次写入日志、通过弹出消息通知用户。



### 设计原则

* 真实性原则：系统内部出现错误，要在用户界面上体现，系统不应该隐瞒运行错误，否则会导致测试成本上升或者交付有隐患的系统。
* 可追溯原则：捕获到的系统运行异常必须如实记录在日志文件中，严禁忽略错误，造成系统上线后的隐患。防止出现系统运行过程中发生了错误，但是日志中没有任何记录情况。
* 最小限度原则：过多的try catch处理会导致程序可读性变差，逻辑不清晰。因此在保证异常得到合理处置的前提下，try catch语句越少越好。
* 分层原则：在设计异常处理方案时，应该对异常进行分类，并在不同的逻辑层次上分类处理。

### 参考设计

#### 异常处理策略

**推荐** 在设计异常处理方案时，推荐采用下面的判定表来决定要不要处理、以及在什么层级上处理，下述的三个层次协同配合，从而保证全部的错误情况都能够得到妥当的处理。

* 第一层 事前检查规避：对于可预见的会引发异常的情况，通过逻辑判断检查识别，从而规避引发异常的情况。比如：在执行除法前检查除数是否为0。
* 第二层 就近捕获：用try语法包围可能引发异常的代码，然后有针对性地根据异常种类不同做不同的处理。
* 第三层 全局捕获：通过注册全局的异常处理器，捕获所有抛出的异常。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 错误原因  处理方式 | 运行环境引发错误 | | | | 非法数据引发的错误 | | 代码有bug |
| 外围系统错误 | | 本系统错误 | | 可预见错误 | 不可预见错误 |
| 调用参数错误 | 其他错误 | 暂时性错误 | 永久性错误 |
| 通过事前检查规避 | **○** | **✕** | **✕** | **✕** | **○** | **✕** | **✕** |
| 全局捕获 | **✕** | **○** | **○** | **○** | **✕** | **○** | **○** |
| 就近捕获 | **✕** | **✕** | **○** | **✕** | **✕** | **✕** | **✕** |
| 写日志 | **✕** | **○** | **○** | **○** | **✕** | **○** | **○** |
| 通知用户 | 告知错误原因 | 告知错误原因 | 告知用户系统忙，请稍后重试 | 请用户联系管理员 | 告知错误原因 | 请用户联系管理员 | 请用户联系管理员 |
| 举例 | 调用支付宝支付，但是支付金额是0 | 调用支付宝支付，返回支付失败。 | 数据库操作超时。 | 数据库连接无效。 | 用户输入的日期不合法 | 外键约束冲突 | 空指针异常、数组下标越界异常等 |
| 备注 |  | 如外围系统不能抛出异常，则应检查返回值后手工抛出异常 |  |  |  | 也可以认为是程序有bug | 修改bug后，转化为可预见的错误。 |

#### 异常处理设计注意事项

软件的异常处理，要防止出现下面几种情况：

* 没有包装的异常

将程序异常直接在界面中弹出，会让用户觉得软件不完善，是半成品。正确的处理方式是将异常详细信息写入日志，用户界面上弹出“系统内部错误，请联系管理员”

* 没有最终捕获的异常

在设计异常处理体系时，一定要保证在最外层拦截所有异常，防止异常导致的程序崩溃。

* 被隐瞒的异常

**禁止** 禁止将异常捕获后不做任何处理。这样会埋下隐患，导致测试过程中无法发现问题。

### 扩展主题

#### 函数的参数检查

**推荐** 一个函数首先应该保证在合法输入的情况下，给出正确的结果。当输入不合法时，函数的行为可以根据不同情况来界定。

对于公共方法，要求做到下图的B模式，也就是当输入不合法时，函数会并且一定会抛出参数异常，这需要编写一定数量的逻辑去检查参数合法性。这样对于公共函数的使用者非常友好，能够节省调试时间。

对于仅在局部使用的私有方法，可以采用A模式，即不检查参数合法性，当输入不合法时，函数的行为不被保证，有可能抛出异常，也有可能输出不确定的结果。这样做的目的是减少不必要的逻辑处理，提高代码的可读性和可维护性，但是会一定程度降低代码的可诊断性。



#### 异常处理与测试

在软件的开发过程中，异常处理并不能一次做好，随着项目的进展，是一个逐渐完善的过程，特别是在测试过程中，还能够发现很多应该事前检查防止的异常、或者需要捕获的异常。

如下图所示，在开发的初期，系统还存在一定的问题和缺陷，但是经过测试，发现了更多需要补充完善的处理，这样就保证了最终软件释放的质量。



为了达到通过测试发现潜在问题的效果，就要求程序的异常处理要严格有序，不能随意捕获后不做任何处理，这样就无法在测试时有效地识别问题。

## 日志处理

### 定义

在程序中写日志是一个非常重要的隐含要求。写好程序的日志可以帮助我们大大减轻后期维护压力。一般需要输出如下种类的程序日志：

|  |  |
| --- | --- |
| 日志种类 | 目的 |
| 异常通知日志 | 用于异常发生时，通知异常监视工具的错误日志。 |
| 异常解析日志 | 与异常通知日志一对一输出的异常解析日志。  在异常通知日志的基础上，输出异常解析用的堆栈信息。 |
| 访问日志 | 记录对本系统的所有访问。  用于监视访问情况以及分析系统的负荷情况等。 |
| SQL日志 | SQL的实行日志。  用于开发时的debug，以及查找有性能问题的SQL。所以正式环境不输出这种日志。 |
| 性能日志 | 性能测试时为了计算性能数据而输出的日志。  正式环境不输出这种日志。 |
| 执行状况日志 | 程序执行状况的日志。  一般输出如下的日志。   * 初期化日志   用于提示日志功能的初期化处理的日志   * 框架输出的日志   框架也会输出一些日志，可据此判断程序的执行状况   * Debug日志   开发时为了Debug而输出的日志 |

### 适用范围

|  |  |  |
| --- | --- | --- |
| 序号 | 项目种类 | 适用情况 |
| 1 | B/S系统 | **○** |
| 2 | C/S系统 | **○** |
| 3 | 移动端APP | **○** |
| 4 | 无界面系统（网关/批处理等） | **○** |
| 5 | 嵌入式系统 | **○** |
| 6 | 维护项目 | **○** |

### 术语表

Log4j：Apache的一个开源项目，提供了一个强有力的日志操作包。

### 区别与联系

日志处理方案与异常处理方案有着密切的关系，需要协同、统筹设计。

### 设计原则

日志处理的设计通常需考虑如下要求。

1. 日志的可读性：日志是给人读的，不仅仅是让自己明白，也要让没有接触过我们源代码的其他程序员也能够一目了然。另外，把日志分类输出到不同的文件也有利于我们排除干扰，迅速找到我们需要的信息。而且，最好在打印日志时输出英文，防止中文不支持而打印出乱码的情况。
2. 日志的性能：无论我们把日志写到文件还是数据库，都需要消耗IO资源。适当的控制日志的输出也有利于提高程序的性能。例如：尽量避免在在大的循环中打印意义不大的日志内容。输出日志之前最好能判断日志的级别(例如. debug前先调用isDebugEnabled()作出判断)。
3. 占用磁盘空间：通常，我们都是把日志写入磁盘上的日志文件中。应考虑适当的使用滚动日志并且定时清除旧文件。
4. 日志的时效性：有的时候我们并不能及时的发现问题。需要追溯之前的日志。所以我们是需要保留一段时间以内的日志便于追溯。

### 参考设计

日志处理的设计通常会考虑以下几个方面，以满足上述的“日志的可读性”、“ 日志的性能”等要求。

* + 日志级别
  + 日志文件的种类和输出设定
  + 日志文件的输出内容
  + 日志处理功能的实现
  + 实现要求

#### 日志级别

日志级别就像开关一样，来决定哪些日志方法被调用，哪些不被调用。通常，日志级别的关系是：TRACE<DEBUG<INFO<WARN<ERROR<FATAL

设置了对应的级别之后，日志框架就只调用大于等于这个级别的方法。

进行日志处理的设计时，需要首先定义如下内容。

* 允许使用的日志级别：如有些系统要求只使用“DEBUG<INFO<WARN<ERROR”这四个级别。
* 各级别日志的使用目的：如“FATAL：发生了导致程序无法继续运行的异常，必须马上对应的情况下使用的日志级别。本日志级别，是异常监视工具的监视对象”。
* 各日志级别的输出对象：定义各日志级别可以在哪个环境里输出。例：

|  | 各日志级别的输出对象 | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| FATAL | ERROR | WARN | INFO | DEBUG | TRACE |
| 正式环境 | ○ | ○ | ○ | ○ | - | - |
| 测试环境 | ○ | ○ | ○ | ○ |  | - |
| 开发环境 | ○ | ○ | ○ | ○ | ○ | - |

#### 日志文件的种类和输出设定

##### 日志文件的种类

需要考虑，使用一个日志文件还是多个日志文件？各日志文件中分别输出什么的日志？使用滚动日志文件时，各日志文件的切换时机是什么？根据环境的不同，上述的问题可能有不同的答案。

* 正式环境的设计例：

| 日志文件名 | 输出的日志种类 | 日志文件切换的时机 | 日志文件的输出单位 |
| --- | --- | --- | --- |
| 异常通知日志文件 | 异常通知日志 | 系统维护时切换 | Server单位 |
| 程序日志文件 | 异常解析日志  访问日志  执行状况日志 | 每天 | 机能单位 |
| 访问日志文件 | 访问日志 | 每天 | 机能单位 |

* 开发环境的设计例：

| 日志文件名 | 输出的日志种类 | 日志文件切换的时机 | 日志文件的输出单位 |
| --- | --- | --- | --- |
| 异常通知日志文件 | 异常通知日志 | 达到指定文件Size时 | Server单位 |
| 程序日志文件 | 异常解析日志  访问日志  SQL日志  执行状况日志 | 达到指定文件Size时 | Server单位 |
| 访问日志文件 | 访问日志 | 达到指定文件Size时 | 机能单位 |

##### 日志文件的切换

需要定义日志文件切换的实现方法。

例：

日志文件的切换由框架自动进行。

切换时机可指定如下。

* 输出文件超过指定的文件size时，自动切换
* 系统日期变更后输出日志时，自动切换

切换后，旧日志文件采用“<原文件名>.yyyyMMddHHmmssSSS.old”的命名规则备份在同一文件夹下。

##### 日志文件的字符编码

需要规定日志文件的字符编码，如“字符编码都使用UTF-8”。

##### 日志文件的输出路径

需要规定正式环境、开发环境等不同环境下各日志文件的输出路径。

#### 日志文件的输出内容

需要设计各种日志的输出时机、输出内容、输出格式等。

以下是“访问日志”的设计例。

访问日志是输出request、response信息。

| No | 输出日志 | 输出时机、输出内容 |
| --- | --- | --- |
| 1 | Request处理开始日志 | 收到Request请求后输出Request内容。 |
| 2 | Request参数日志 | 输出收到的Request参数的信息。 |
| 3 | Dispatch日志 | 输出处理本Request的Action名。 |
| 4 | Request处理终了日志 | Server端处理终了时输出Server端的处理结果。 |

Request处理开始日志的format

@@@@ BEGIN @@@@ rid = [<Request ID>] uid = [<User ID>] sid = [<Session ID>]

url = [<URL>]

method = [<HTTP Method>]

port = [<port>]

client\_ip = [<Client IP Address>]

client\_host = [<クライアント端末ホスト>]

Request参数日志的format

……

Dispatch日志的format

……

Request处理终了日志的format

……

Request处理开始日志的输出例

2012-08-23 18:48:06.143 -INFO- ACC [201208231848061430029] @@@@ BEGIN @@@@ rid = [RWXX0001] uid = [9999999999] sid = [1B0F055D05314CA0753180C42B02ECDB]

url = [http://localhost:8880/action/AABB/WXX01Action/RWXX0101]

method = [POST]

port = [8880]

client\_ip = [127.0.0.1]

client\_host = [127.0.0.1]

Request参数日志的输出例

……

Dispatch日志的输出例

……

Request处理终了日志的输出例

……

#### 日志处理功能的实现

需要确定日志处理功能的实现方法。有些框架提供了自己的日志输出方法，否则可以使用Log4j。

无论采用哪种实现方法，都应该明确规定配置文件的设定内容和日志输出代码的示例，以保证系统中日志处理功能的一致性。

#### 实现要求

关于日志处理的实现要求或禁止事项，也应在设计时列出。

例如，关于debug目的的日志输出实现，可列出如下要求。

为了开发测试时的DEBUG而输出日志的时候，需遵循以下的实现方针。

* 必须用DEBUG、TRACE这种日志级别（这种级别的日志在正式环境中不被输出）。
* 每个需输出日志的class都需从LoggerManager取得logger，并保存在class变量里。
* 需进行message组合并输出时，应判断当前日志级别后再进行message的组合及输出处理，以免在正式环境中不需输出这些日志时，还进行不必要的组合处理导致性能劣化。

//取得logger，并保存在class变量里

private static final Logger debugLogger = Logger.get(UserManager.class);

//判断日志级别后进行输出处理

if (debugLogger.isDebugEnabled()) {

String message = "loginId[" + loginUser.getId() + "],loginName[" + loginUser.getName() + "]";

debugLogger.debug(message);

}

## 配置文件

### 定义

配置文件是指用来更改应用设置并为其设置策略的环境设置和文件集合。配置文件的应用场景繁多，从底层的操作系统到各种中间件，再到各种应用软件，都有其身影。配置的形式也多种多样，包括环境变量、数据库配置、文件配置等。我们主要讨论应用软件层面的常用配置文件。

按照配置文件的不同作用，配置文件可分为：应用程序配置文件(application)、日志配置文件(log)、db配置文件(db)、消息配置文件(message)等；按照配置文件的类型，配置文件可分为：properties文件、xml文件、ini文件、yml文件等。

|  |  |  |
| --- | --- | --- |
| 类型(不同作用) | 常见作用 | 备注 |
| 应用程序配置文件(application) | 应用的启动方式  应用的加载项目  应用展示的文字  应用的特殊行为 | 启动模式(testModal=true)  启用哪些功能  不同环境/用户显示不同文字  应用内业务使用配置项 |
| 日志配置文件(log) | 日志是否输出  日志的输出级别  日志的输出位置  日志的输出格式 | 标准日志框架配置 |
| db配置文件(db) | 连接哪个db  连接的用户名  连接的密码  连接的方式 | 标准db连接配置 |
| 消息配置文件(message) | 消息内容 | 提示内容 |

### 适用范围

|  |  |  |
| --- | --- | --- |
| 序号 | 项目种类 | 适用情况 |
| 1 | B/S系统 | **○** |
| 2 | C/S系统 | **○** |
| 3 | 移动端APP | **△** |
| 4 | 无界面系统（网关/批处理等） | **△** |
| 5 | 嵌入式系统 | **△** |
| 6 | 维护项目 | **✕** |
|  |  |  |

### 术语表

无。

### 区别与联系

配置文件的文件类型多样，但所起的作用都是一样的，都是用来更改应用设置并为其设置策略的文件的集合。

不同类型的配置文件都有其常见适用的领域和优缺点。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型(文件类型) | Properties文件 | ini文件 | Xml文件 | Yml文件 |
| 应用领域 | Java系列 | Windows/微软系列 | 几乎所有语言和系统都支持 | Spring boot/运维等脚本语言(python等) |
| 注释符 | # | #和; | <!-- 🡪 | # |
| 特点和优点 | 键值对  简单 | 扩展的properties，支撑用[section]来进行分层 | 1. 支持复杂的数据结构 2. 层级关系 3. 易理解互操作性强 4. 支持多种编码 5. 规范统一 | 1.大下写敏感  2.使用缩进表示层级关系  3.缩进时只允许空格，不允许tab  4.缩进的空格不重要，只要相同层级的左侧对其即可  5.与xml比  简洁、  和脚本语言交互良好、易于实现等 |
| 缺点 | 不支持层级关系 | 1. 结构只有2层 2. 有64k限制 3. 复杂对象描述比较乏力 | 解析速度慢  占用更多存储空间 |  |
| 适用于 | 配置内容简单少,无层级  只要支持读写properties的所有语言都适用 | 配置内容简单少，1个层级  只要支持读写ini的所有语言都适用 | 配置内容复杂多，  大型系统，  和其它系统的交互配置 | 运维配置，  大型系统中原生框架自带yml配置的场合 |

### 设计原则

1. 统一性原则：所有配置内容统一规范设计。
2. 可管理原则：系统中的所有配置内容，必须使用公共类(方法)进行读写。
3. 易理解原则：配置文件的key和value应该符合人们对该配置项的客观认识，能够一目了然的知道该配置项代表的含义。
4. 适合性原则:根据配置项多少和配置内容的层级复杂度，选择适合的配置文件。

### 参考设计

通常，一个系统的配置文件设计需要考虑以下几个方面：

* 什么场合需要使用配置文件
* 配置文件的设计规范和内容
* 配置文件的管理和读写
* 是配置文件还是硬编码

#### 配置文件使用的场合

在进行设计时，需要结合当前系统的需求和特点，确定是否需要使用配置文件。下述内容为一般情况的设计和定义，供参考。

* 当存在系统在运行过程中，想不重新打包编译，就能够改变系统的某些行为或者表示的情景时，就需要考虑使用配置文件。如程序有异常了，想调整一下出力log的级别，方便发现问题，就可以通过修改日志配置文件的日志输出级别，不重新编译发布即可达到目的。
* 开发环境和发布环境不一致这种环境不一致的情况，也可以通过配置文件来无缝切换。如使用spring boot的时候，通常我们就会有application-dev.properties和application-deploy.properties的配置文件，里面分别配置开发和发布的环境设置，在application.properties中指定生效的配置文件，达到无缝切换开发和发布环境的目的。
* 系统支持国际化的时候，一般所有的表示项，都需要配置在配置文件中。针对不同的语言，有不同语言的配置文件。例如资源文件，形如message\_language\_country.properties，

英文配置message\_en\_US.properties、中文配置message\_zh\_CN.properties。

* 不确定的场合。根据业务不同，会有一些不确定的因素，我们一般也通过配置文件来达成目的。如某一个业务操作不确定，其处理就可以先使用抽象类和一个默认实现类，等到业务明确后，修改配置的处理类，让其执行特定的业务处理。

上述各种需要配置文件的场合，如下表

|  |  |
| --- | --- |
| 场合 | 是否需要配置文件 |
| 动态行为和显示 | ○ |
| 配置环境 | ○ |
| 国际化 | ○ |
| 不确定的处理 | ○ |

#### 配置文件的设计规范和内容

配置文件的内容应该有统一的设计规范。

配置编码是指为系统中的全部配置编制统一规则的代码，代码和配置项一一对应。配置代码除了在程序中作为常量名称指代配置项值以外，开发人员还可以根据配置代码来推测该配置项的作用和应用范围等信息。因此，配置代码易读是有必要的，有利于大家的使用。下面以.properties配置文件为例，进行说明。

**推荐** 以下的配置代码设计可供在项目中参考。

|  |  |  |
| --- | --- | --- |
| 配置文件的功能代码  （n位字母） | 功能内代码  （n位字母） | 结合key |
| 如db相关配置，可以使用db  应用系统设置，可以使用application  消息设置可以使用  message | 功能内代码贴合其实际含义为准  例如db的连接用户和密码  username  password | db.username  db.password |

在定义配置编码时，没有特殊情况建议都使用英文单词全拼作为编码：

* 如用户名，使用username，不用用u;密码使用password，不要使用p
* 特殊的，如数据库database，业界通用的都是db，这种业界统一规范，遵循业界规范

#### 配置文件的管理和读写

**强制** 为了保证系统行为的一致性，避免开发人员随意定义和使用配置文件引起的诸多问题，提高可维护性，要求系统中所有配置文件必须统一管理由专人维护并且实现统一的配置文件读写公共方法。所有配置文件都由一个文档或文档目录管理，代码读写必须经由统一的公共方法。

**推荐**系统配置文件，建议按照功能的不同，分为多个文件。

如果每个文件的内容都比较少(1-2条)，建议合并。

**推荐** 配置文件读写的公共方法参数设计参考下表：

读取配置文件(getProperty)

|  |  |  |
| --- | --- | --- |
| 序号 | 参数 | 说明 |
| 1 | key | 配置文件的key |
| 2 | defaultValue | 如果没有读取到的默认值 |

**不推荐**单个系统的配置文件，不推荐系统进行写操作。

写配置文件(setProperty)

|  |  |  |
| --- | --- | --- |
| 序号 | 参数 | 说明 |
| 1 | key | 配置文件的key |
| 2 | value | 配置文件的内容 |

##### Spring boot读取示例

以读取properties文件为例

Spring boot中读取properties文件常用3种方式

config.properties文件中有配置项目

db.username=test

db.password=test

方式1：使用Environment

能够按照键值读取所有properties文件中的内容

@Autowired

private Environment environment;

获取配置

public String getDbUsername() {

return environment.getProperty (“db.username”);

}

// 如果读取不到配置，返回传入的默认值

public String getDbUsername(String defaultUsername) {

return environment.getProperty (“db.username”, defaultUsername);

}

方式2：使用@Value注解

指定properties文件中的键

@Value(“db.username”)

private String username;

带默认值的配置(db.username2键不存在,返回not found username2)

@Value(“db.username2:not found username2”)

private String username2;

方式3：使用@PropertySource@ConfigurationProperties注解，结合

指定指定properties文件中指定前缀的内容，自动装配config.properties中db.开头的配置文件到对应的username和password属性中。

@Component

@PropertySource(value="classpath:config.properties")

@ConfigurationProperties(prefix="db")

public class ConfigProperties {

private String username;

private String password;

}

##### 配置的更新

在应用运行过程中，通过改变配置动态改变应用的行为的前提是，配置在应用运行过程中能够动态加载并应用。这个要求应用程序本身具有重新加载配置的能力。

**注意**应用重新加载配置有2点需要注意：

1. 应用重新加载配置，不能导致应用终止。
2. 一般为了提升性能，配置都是会被应用缓存的，针对缓存的配置需要考虑刷新配置的时机和刷新配置事务的完整性。

配置更新，配置更新可以直接更新配置文件。也可以通过应用提供的配置更新接口更新配置。

例如spring系列就提供了远程更新配置的能力。如果有需要，请参考如下链接：

(<https://blog.csdn.net/lblblblblzdx/article/details/81784237>)

**建议**如果有能力，可以采用配置中心统一管理配置，配置中心提供统一的配置维护功能。

#### 是配置文件还是硬编码

**推荐**不变的可以选择硬编码或者配置文件。可变的一般是配置文件，如果配置文件修改代码也必须修改的，这个情况硬编码和配置文件都可以。例如支撑的业务类型扩展或者变更，代码必须改变的。

硬编码的好处，代码中易于引用，不用再从配置文件读取一次。

**推荐**实际应用中，建议一个配置文件对应一个配置类，配置类中的变量和配置文件中的配置项一一对应，这样就兼顾了配置的使用便利。

### 参考资料

# 实现设计(Design Definition)-数据访问域

## 排他设计

## 数据库连接策略

## 事务处理

### 定义

事务：要么什么都不做，要么做全套(All or nothing)。事务是恢复和并发执行的基本单位。在关系数据库中，一个事务可以使一条SQL语句，一组SQL语句或整个程序。事务的核心是锁和并发，保证只有一个线程进入到一个事务中。

#### 事务特性

事务应具有4个特性(ACID)：原子性(Atomicity)、一致性(Correspondence)、隔离性(Isolation)和持久性(Durability)。

4个特性如下：

|  |  |  |
| --- | --- | --- |
| 事务特性 | 说明 | 备注 |
| 原子性(Atomicity) | 一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做 |  |
| 一致性(Correspondence) | 事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。 |  |
| 隔离性(Isolation) | 一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。 |  |
| 持久性(Durability) | 持久性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。 |  |

#### 事务隔离级别

ANSI/ISO SQL定义的数据库事务的隔离级别有4个，由低到高依次为未提交读(read uncommitted)、提交读(read committed)、重复读(repeatable)、可串行化(Serializable)。

事务隔离级别

|  |  |  |
| --- | --- | --- |
| 隔离级别 | 说明 | 备注 |
| 未提交读(read uncommitted) | 所有事务都可以看到没有提交事务的数据 | 会出现脏读、不推荐 |
| 提交读(read committed) | 事务成功提交后才可以被查询到。 | 会出现不可重复读、推荐 |
| 重复读(repeatable) | 幻读，事务A读取数据，锁定数据记录，其他事务不能修改被事务A锁定的记录，但可以插入新的数据。 | 会出现幻读 |
| 可串行化(Serializable) | 强制的进行排序，在每个读读数据行上添加共享锁 | **会导致大量超时现象和锁竞争，没有特殊情况，不推荐。** |

隔离级别和问题关系如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 隔离级别 | 脏读 | 不可重复读 | 幻读 |
| 读未提交 | 可能 | 可能 | 可能 |
| 读已提交 | 不可能 | 可能 | 可能 |
| 可重复读 | 不可能 | 不可能 | 可能 |
| 可串行化 | 不可能 | 不可能 | 不可能 |

#### 事务传播行为

传播行为分为支持事务的传播和不支持事务的传播。

传播行为如下

|  |  |  |
| --- | --- | --- |
| 传播行为 | 说明 | 是否支持事务 |
| PROPAGATION\_REQUIRED | 如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。 | 支持 |
| PROPAGATION\_SUPPORTS | 支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。 | 支持 |
| PROPAGATION\_MANDATORY | 支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。 | 支持 |
| PROPAGATION\_REQUIRES\_NEW | 创建新事务，无论当前存不存在事务，都创建新事务。 | 支持 |
| PROPAGATION\_NOT\_SUPPORTED | 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。 | 不支持 |
| PROPAGATION\_NEVER | 以非事务方式执行，如果当前存在事务，则抛出异常。 | 不支持 |
| PROPAGATION\_NESTED | 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。 | 不支持 |

### 适用范围

|  |  |  |
| --- | --- | --- |
| 序号 | 项目种类 | 适用情况 |
| 1 | B/S系统 | **○** |
| 2 | C/S系统 | **○** |
| 3 | 移动端APP | **△** |
| 4 | 无界面系统（网关/批处理等） | **△** |
| 5 | 嵌入式系统 | **△** |
| 6 | 维护项目 | **✕** |
|  |  |  |

### 术语表

无。

### 单机事务

单机事务一般指数据库事务。对应数据库的事务隔离级别和传播行为。Java中对应的JDBC事务。

### 分布式事务

分布式事务就是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。

分布式事务的分为传统的分布式事务和互联网分布式事务。传统的分布式事务，一般采用分阶段提交的方式实现，互联网的分布式事务追加的是最终一致性。

**推荐**对于强业务一致性，建议利用各个数据库的二阶段提交的方式实现(要求数据库本身支持二阶段提交)。Java中对应的JTA事务(XA协议:二阶段提交协议)。

互联网的分布式事务实现比较复杂，通常根据业务场景来考虑。本文不展开论述。

### 各种数据库的默认和推荐事务级别

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 数据库 | 默认级别 | 推荐级别 | 备注 |
| 1 | Oracle | 读已提交 (READ COMMITTED) | **默认级别** |  |
| 2 | Postgresql | 读已提交 (READ COMMITTED) | **默认级别** |  |
| 3 | mysql | 可重复度(REPEATABLE-READ) | **默认级别** |  |
| 4 | Sql server | 读已提交 (READ COMMITTED) | **默认级别** |  |
| 5 | Db2 | 读已提交 (READ COMMITTED) | **默认级别** |  |
| 6 | Mongdb(Ver>=4.0) |  |  | 4.0以上支持事务 |
| 7 | redis |  |  | 失败不回滚 |
| 8 | DynamoDB |  |  | 提供事务支持 |

**推荐**关系数据库的默认隔离级别推荐为读已提交(READ COMMITTED)

### 事务隔离级别调整原则

1. 默认原则：事务隔离级别，通常使用我们选用的数据库的默认级别即可。各种数据库的默认级别，见各种数据库默认级别表。
2. 调整原则：特殊业务需要的时候，选择性调整数据库事务。

如某个系统要求更新性能尽可能高，不关心是否出现脏读的情况，这个时候，就可以把数据库事务调整为未提交读(read uncommitted)。(性能为什么会提升呢？通常数据库实现事务隔离级别都是通过加锁来实现的，当调低隔离级别后，加的锁就少了，同一时刻对临界资源的抢占概率就降低了，自然性能就上去了)

如某个系统要求不能出现脏读、重复度、幻读的场合，这个时候，就可以把数据库事务隔离级别调整为可串行化(Serializable)。

**注意**调整数据库隔离级别的时候，需要考虑对应的数据库是否支持该隔离级别。如oracle，就只支持提交读(read committed)和可串行化(Serializable)，其它隔离级别不支持。

### 事务设计要点(AOP)

AOP面向切面编程。通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。利用AOP可以帮助大家把业务逻辑各个部分进行分离，降低业务逻辑各部分的耦合度，提高程序的可重用性，提高开发效率。

AOP通常可以用来分离事务、日志、性能统计、安全控制、异常处理等系统级服务，让大家只关注核心业务。

基于AOP管理事务需要考虑以下几个方面：

* 切入点(事务开启和结束的层次，在Controller还是Service还是dao层开启事务)
* 哪些处理需要事务(拦截的方法名)
* 提交和回滚

#### 切入点

单一架构应用，建议事务的切入点放到业务层(Service),

事务的传播行为适用默认的级别(PROPAGATION\_REQUIRED)(如果当前没有事务,就创建一个事务,如果当前已有事务，就加入该事务)

各个业务有聚合的情况，建议再增加一个接口Service，在该Service中调用各个业务Service，搭配默认的事务传播行为。

#### 哪些方法需要事务

如果该方法，有DB更新处理(插入、更新、删除、或者调用存储过程中存在更新行为)且该方法处理参与的一次业务流过程中存在db更新操作 >= 2次的情况，才建议使用事务管理。

如果该方法只是查询，不用添加事务处理。

如果该方法参与的所有业务流过程中不存在超过1次的DB更新操作，也不用添加事务处理。

#### 提交和回滚

提交：当某次处理没有预定义回滚中的异常发生时，执行完切面方法后，事务提交。

回滚：当某次护理发生回滚中定义的异常时，切面捕获到后，进行事务回滚。

#### Spring 对事务的支持

Spring支持编程式事务和声明式事务两种管理方式。推荐使用声明式事务管理，因为声明式事务管理不需要在业务逻辑代码中掺杂事务管理代码，是非侵入式的开发方式。声明式事务只能作用到方法级别，不能和编程式事务一样能够作用代码块，这一点是其相对于编程式事务的不足。

#### Spring AOP

Spring的Aop动态代理支持两种形式，jdk动态代理和cglib的动态代理。

**注意**

1.jdk动态代理是基于接口实现的，这要求代理的类都是面向接口的，代理的方法是接口中定义的。

2.Cglib的动态代理是基于继承类实现子类的方式实现的，要求切入的类和方法不能是final修饰的。

### 参考资料

# 实现设计(Design Definition)-安全域

## 用户认证方案

## 角色权限方案

## 敏感信息加密方案（完善中）

### 定义

敏感信息是指不当使用或未经授权被人接触或修改后，会不利于国家和组织利益或不利于个人依法享有的个人隐私权的所有信息。根据其信息种类的不同，可以分为个人敏感信息、商业敏感信息、国家秘密。

敏感信息加密是指通过加密算法对敏感信息进行编码，使其改变原有的形式，从而无法被直接读取识别，只有通过特定的算法才能将数据还原为原有的形式的一系列过程。

它可以应用于少量数据，如密码、邮箱地址、电话号码等，也可以文件，甚至是存储介质中保存的大量数据。

### 适用范围

|  |  |  |
| --- | --- | --- |
| 序号 | 项目种类 | 适用情况 |
| 1 | B/S系统 | **○** |
| 2 | C/S系统 | **○** |
| 3 | 移动端APP | **○** |
| 4 | 无界面系统（网关/批处理等） | **○** |
| 5 | 嵌入式系统 | **○** |
| 6 | 维护项目 | **✕** |

### 术语表

对称加密算法：指加密和解密使用相同密钥的加密算法。常见的对称加密算法有DES、AES等。优点是加密速度快，缺点是安全性相对低。

非对称加密算法：指加密和解密使用不同密钥的加密算法，又称公钥加密算法。常见的非对称加密算法有RSA、EIGamal等。优点是安全性高，缺点是加密速度慢。

数字摘要：是采用单项Hash函数将需要加密的明文“摘要”成一串固定长度的密文，这一串密文又称为数字指纹，它有固定的长度，而且不同的明文摘要成密文，其结果总是不同的，同样的明文其摘要必定一致。通常用来确保数据完整性和防止数据篡改。

数字签名：是结合“非对称密钥加解密”和“数字摘要“技术，通过对比接收到的报文的数字摘要与接收到的数字摘要的明文是否一致，来确认报文是否是从发送方发送过来且没有被篡改过的一项技术。

数字证书：是由CA机构发行的一种电子文档。该证书内包含证书申请用户的个人信息和公钥信息，同时还附有认证中心的签名信息。通过权威的值得信赖的第三方CA机构来保证公钥的真实可靠。

加密模式：描述是把明文按照某种单位进行分解，通常是按块单位进行分解，然后对分解的各部分分别按某种规则进行加密的过程。如电子密码本模式、密码块链模式等。

数据块填充模式：大多数密码算法都是块密码算法，需要将明文消息切成固定大小的块，一块一块地进行加密。当明文长度不是块大小的整数倍时，需要按照某种方式对明文进行填充。常见的填充模式有Zeros填充、X923 填充、PKCS7 填充、ISO10126 填充。

加密强度：用来衡量密码算法或密码系统的安全性，对破解密码算法或系统所需要工作量的一个数值度量。强度越高，破解所需付出的代价就越大。

### 区别和联系

信息脱敏和敏感信息加解密是两种不同的技术，适用于不同目的。

信息脱敏一般来说是按照某种既定规则对数据进行变形，只要处理到无法推断原有的信息，不会造成信息泄露即可。往往不需要保证信息的可恢复性。

### 设计原则

* 高效性原则：根据加密对象的不同，合理的选择加密算法，保证加密过程的性能。通常，加密对象内容较多时，选择对称加密，加密对象内容较少，且对安全性有较高要求是，选择非对称加密。
* 高加密强度原则：在加密性能允许的前提下，尽可能选择安全性高、加密强度的加密算法或多种加密算法的组合。例如，对于对称密钥进行加密时，因为对称密钥本身的长度较小，尽量选择非对称加密算法。

### 参考设计

通常，一个系统的敏感信息设计需要考虑以下集中场景：

* URL中的敏感信息加密
* 基于网络传输的敏感信息加密

#### URL中的敏感信息加密

##### BASE64编码

**推荐**为了防止重放攻击，把URL加上一个随机数，然后进行BASE64编码。

##### 加密请求报文

**推荐**因为BASE64本身并不是一种加密技术，为了保证安全性，需要选择对称加密算法对编码后的URL进行加密。

#### 基于网络传输的敏感信息加密

##### 生成签名

为了验证请求报文的内容未被篡改，需要对请求报文进行数字签名处理。

**推荐**这时要首先使用Hash算法生成请求报文的摘要，然后使用非对称加密算法对摘要进行加密。

##### 加密请求报文

**推荐**当需要对请求报文中包含敏感信息，需要进行加密传输的时候，为了保证加密性能，需要选择对称加密算法进行加密。

##### 对称密钥加密

**推荐**在对请求报文内容进行对称加密后，为了保证对称密钥在网络传输中的安全性，需要使用非对称加密算法对对称密钥进行加密。

##### 申请数字证书

**推荐**为了证明身份和公钥的真实可靠，向CA机构申请证书。

## 恶意攻击预防

# 系统分析(System Analysis)

## 关键指标验证

### 并发量指标

#### 并发测试的关键因素

多用户和同时性是并发测试中的两个关键因素，两个因素缺一不可，只有多用户但是不同时操作，很难对系统构成压力，没有多用户，同时的概念也就不存在了。

并发用户数的计算：在一定的时间范围内，平均同时在线用户数量常用公式：C=N\*L/T，

C是平均的并发用户数；n是login session的数量；L是login session的平均长度；T指考察的时间段长度。比如，上班签到系统，早上8：30上班，8点到8：30的时间里员工会登录签到系统进行签到。公司员工为1000人，平均每个员上登录签到系统的时长为5分钟。可以用下面的方法计算。C=1000\*5/30=166.7，C表示平均并发用户数，那么对这个签到系统每秒的平均并发用户数为166。

最大并发数C’可以用平均并发数进行推算，常用的推算公式为：

#### 并发压力测试的典型策略

* 独立业务性能测试

核心业务模块某一业务并发性能测试，独立业务性能测试主要有以下类别

完全一样功能的并发测试：检查程序对同一时刻并发操作的处理，例如模拟多个用户在同一时刻向数据库写入相同数据，或者多个用户在同一时刻发出请求测试系统能否正确响应。

完全一样操作的并发测试：在同一时刻完成完全一样的操作，即从宏观上看操作对系统的影响是一致的，例如同时单击保存按钮。这类测试目的在于验证大量用户使用同一功能时系统能否正常工作。

相同不同的子功能并发测试：同一模块大多数功能相互耦合，针对一些子功能较多的模块做组合测试。组合的依据就是用户使用的场景，每个不同的子功能都模拟一定的用户数量进行并发测试。例如，同一时刻，有的用户进行登录，有的提交表单。

* 组合业务性能测试

一个或多个业务模块的多个业务同时进行的并发测试，组合业务性能测试主要有以下类别

不同核心业务模块的用户进行并发，模块之间具有一定耦合：这种测试比较接近用户使用情况，测试的对象是多个模块组，每个组相关的模块之间具有一定耦合关系。组与组之间的关系相对独立。例如实际中各类型的用户都会对应一组模块，相当于不同的业务组并发的访问系统。

具有耦合关系的核心模块组进行并发，每组模块内部存在耦合关系：主要测试多用户并发条件下一些存在耦合或者数据接口的模块是否正常运行，可以参考集成测试用例和概要设计文档，分析出一些核心模块的接口。

基于用户场景的并发测试：选择用户的一些经典场景做测试，测试对象可以使核心模块，也可以是非核心模块。这种测试更接近用户使用的实际情况，测试需要充分考虑实际场景。设计组合模块用户并发性测试用例一般用不同“子功能”或者“子事务”为单位来进行各个模块的不同核心功能组合。

#### 并发[性能测试流程](https://www.cnblogs.com/puresoul/p/5456855.html)

##### 测试流程

√需求分析

√测试计划指定与评审

√测试用例设计开发

√测试执行与监控

√分析测试结果

√编写性能测试报告

√测试经验总结

##### 并发测试用例设计

并发测试用例设计中关键的是并发用户数量设计，其中几个比较关键的概念

一、并发用户：

多个用户在同一时间做同一操作；

多个用户在同一时间向系统发出请求（可以相同也可以不同）

二、并发用户数量：

在同一时刻与服务器发生交互的在线用户数量，这种交互可以使单向传送数据包也可以是双向传送数据包。并发用户数量设计方法一般有三种

1.极限法：

取最大在线用户数当做最大并发数。这种方法适用于已经投产或者使用目标不明确的门户网站，也可以使用已经注册的用户数做系统用户数量数，计算最大并发数量（经验公式不确定，根据系统类型，项目经验确定，例如办公系统：使用系统的用户数量

2.用户趋势分析：

对系统生存周期内的用户未来走势进行分析，预测系统未来可能达到的最大使用数目，从而估计最大并发用户数，这种方法多用于用户不断增加的情况。

3.经验评估法：

根据经验判断最大并发用户数，这种方法适用于用户数量相对稳定且明确的系统。具体项目中通常几种方法结合使用来确定最大并发用户数量。完成最大并发用户数量评估后按照其百分比来设计每个测试用例需要模拟的用户数量，例如可以按照最大并发用户数量的不断增加来设计模拟用户数量，直到达到最大并发用户数量。对于某一特定用例，设计用户数量需要注意：

按照各类用户同时递增的方式来设计用户数量。按照递增顺序设计用例是为了按照由浅入深的来发现系统瓶颈。

并发用户数的最大值一般不超过最大并发用户数量的20%，除非是为了测试系统能支持的最大并发用户数量。

设计用户数量时要考虑成本，因为每组用户都意味着至少一次测试。系统不同时间段场景设计不同时间段的场景设计更接近用户使用情况，是核心模块和组合模块并发性能测试用例的基础。

##### 性能测试的步骤

性能测试一般的步骤包括以下几个部分，

* 1. 搭建与维护测试环境
  2. 执行测试用例
  3. 监控测试执行场景
  4. 保存与分析测试结果

当测试环境准备好之后就可以执行测试用例，测试执行大多借助工具完成。

#### 常用并发性能测试工具

* + Postman

Postman一款非常流行的API调试工具。其实，开发人员用的更多。因为测试人员做接口测试会有更多选择，例如Jmeter、soapUI等。不过，对于开发过程中去调试接口，Postman确实足够的简单方便，而且功能强大。

Postman常用于接口的测试，比如GET，POST等接口的测试

* + Apache Bench

ApacheBench 是 Apache 服务器自带的一个web压力测试工具，简称ab。ab又是一个命令行工具，对发起负载的本机要求很低，根据ab命令可以创建很多的并发访问线程，模拟多个访问者同时对某一URL地址进行访问，因此可以用来测试目标服务器的负载压力。总的来说ab工具小巧简单，上手学习较快，可以提供需要的基本性能指标，但是没有图形化结果，不能监控

* + JMeter

Apache JMeter是Apache组织开发的基于Java的压力测试工具。用于对软件做压力测试，它最初被设计用于Web应用测试但后来扩展到其他测试领域。它可以用于测试静态和动态资源例如静态文件、Java 小服务程序、CGI 脚本、Java 对象、数据库， FTP 服务器等等。JMeter 可以用于对服务器、网络或对象模拟巨大的负载，来在不同压力类别下测试它们的强度和分析整体性能。另外，JMeter能够对应用程序做功能/回归测试，通过创建带有断言的脚本来验证你的程序返回了你期望的结果.

### 数据库处理

### 响应时间

### 批处理时间

## 外部系统调用与数据交换

### 认证机制

### 时序机制

### 数据格式

### 数据交换文件

# 程序构建(Implementation)

## 开发环境搭建

### 代码版本的搭建和使用

#### 开发环境搭建的目标和范围

开发环境的构建是开发的前提条件，整个开发环境会影响到很多问题：开发效率、开发版本正确性、因环境导致bug数量、软件测试版本和发布版本的准确性、和环境相关问题发现的时机点等等。

所以环境搭建要考虑如下几个目标：

1. 公开性 ： 整个开发环境的信息必须是公开的、共享的、全员必须清楚明了的。所以需要考虑公开相关软件、第三方lib包等，建立并公开环境清单一览、环境构建说明书等文档。
2. 简易性： 整个开发环境的搭建需要简单方便，不能出现太过复杂的安装过程。如果环境搭建过于复杂会影响开发的效率。
3. 集成性： 为了实现开发环境搭建的简易性，可以考虑尽可能多的集成开发软件。比如以java开发为例，可以将 （eclipse + jdk + maven + tomcat + 工作空间配置） 尽可能多的直接集成在一个文件夹中，然后全员公布
4. 一致性：开发环境和测试环境要保证全员一致。如果在不一致的情况下可能会产生bug的误报和无意义的调查时间浪费。
5. 易升级/回滚： 能够方便升级或者进行回滚操作。比如出现第三lib版本升级，可以快速的全员获得新版本并完成升级。比如代码出现严重错误，全员可以快速的回滚上一个版本节点
6. 模拟性： 大量软件开发过程中需要引用第三方接口、第三方软件、文件服务、邮箱服务等其他环境。所以在环境搭建过程中也是一个不可缺少部分。要100%模拟其他外部环境才能在开发和测试的过程中保证开发作业的完整性，并降低bug。
7. 测试性：开发的下一个阶段的是发布测试、所以开发环境运行的程序必须具有测试的准确性。所以开发环境的搭建要和测试环境乃至生产环境保持一致，才能最大降低在测试环境下产生BUG数量。
8. 发布性：从代码打包为软件是在提测阶段和发布阶段重要的一个流程节点，所以应该考虑如何从代码版本打包是正确的方式。并且应在程序中构建多环境配置文件，能够在各个阶段不修改任何代码即可快速打包。
9. 容灾性：开发环境应考虑故障灾害反生的处理策略。所以在开发环境中，保证最新文件数据应在服务器上，请需要定期备份。防止灾害反生时，导致全项目组工作暂停的事情发生。

#### 开发环境搭建的范围

开发环境搭建的范围：

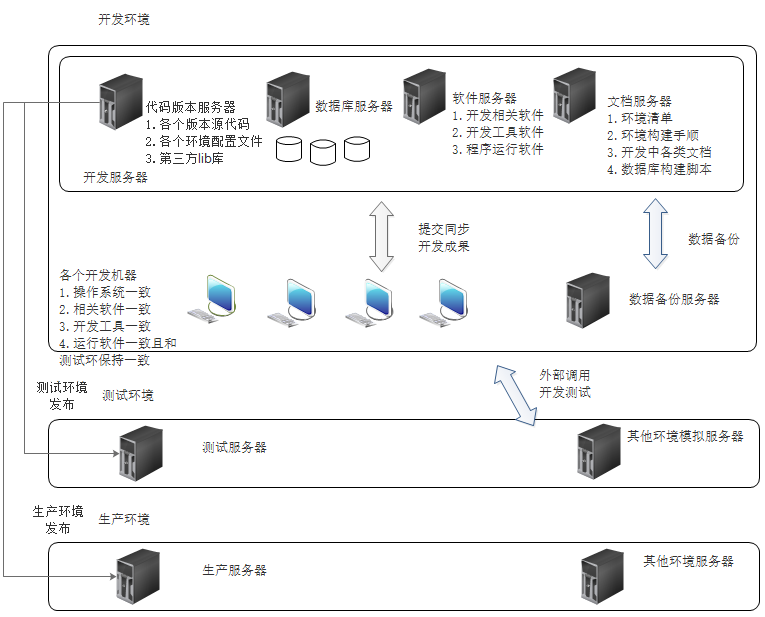
1. 操作系统：操作系统是开发环境的基础。开发之初要确定开发机、服务器和测试服务器的操作系统类型版本等
2. 开发相关软件：开发相关软件指的是完成开发工作之外需要使用的软件。比如：提交代码需要使用svn客户端软件。
3. 开发工具软件：开发工具就是开发的IDE软件。比如数据库客户端、java开发工具Jdk、Eclipse等。
4. 运行工具软件：运行工具软件指的是程序运行所需要的软件。比如Java开发中jre、tomcat服务器等。
5. 开发服务器：开发中需要使用的文档服务器、软件仓库服务器、数据库服务器、代码版本控制服务器及备份服务器。要确定类型和数量。
6. 模拟服务器：模拟开发过程中外调其他接口、远程文件操作等操作的服务器

在整个开发环境搭建准备初期，建立一张开发环境软件和硬件的清单是有助于我们分析和构建开发环境。下面以一个java语言开发的web系统为例，展现一个开发环境的清单。

开发环境清单一览示例：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 环境种类 | 名称 | 数量 | 操作系统 | 软件及版本 |
| 开发环境 | 代码版本服务器 | 1 | windows server 2008 r2 enterprise 64位 | VisualSVN-Server-3.5.0-x64 |
|  | 数据库服务器 | 1 | Centos7.4 64位 | Mysql-8.0.16-winx64社区版 |
|  | 开发机器 | 5 | Win10 专业版 | 相关软件：  TortoiseSVN\_1.8.11.26392\_x64  开发工具：  jdk1.8.0\_77 x64  eclipse 4.6.0 x64  运行软件：  Apache Tomcat 7.0.57 |
| 测试环境 | web服务器 | 1 | Centos7.4 64位 | jdk1.8.0\_77 x64  nginx-1.14.2  Apache Tomcat 7.0.57 |
|  | 数据库服务器 | 1 | Centos7.4 64位 | Mysql-8.0.16-winx64社区版 |
|  |  |  |  |  |

#### 开发环境搭建的总体架构图



### 代码和数据库脚本的版本控制

#### 代码和数据库脚本的版本控制

开发中如果没有版本控制的话，代码可能被别人或自己不小心覆盖或遗失、也不知道是谁因为什么原因改了这段代码、也没办法可以复原回前几天的修改。

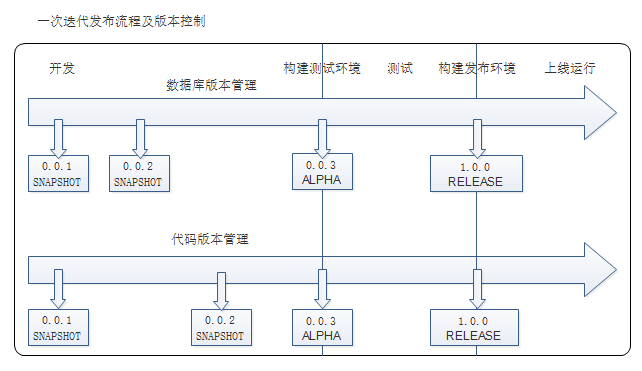
有了版本控制，可以浏览所有提交历史纪录，变更差异，掌握团队的开发进度，任何修改都增加了保障性，也可快速的回滚上一个版本。

分支和标签的功能可以软件发行的不同版本，例如稳定版本、维护版本和开发中版本。可以促进团队的快速迭代。让团队完成更复杂的迭代任务，一条分支修改现有版本产品的bug，一条分支进行产品升级。

同样数据库脚本的版本控制除了有以上的好处之外，可以让我们快速的构建新的正确的数据库开发环境或者测试环境。在开发过程中代码的版本控制和数据库脚本的版本控制是需要同步进行的。特别是在一些变更频繁或者升级频发的项目中，使用数据库脚本版本控制的意义尤为重大。新的数据库脚本版本可以快速应用到每个开发数据中，让开发人员在正确的版本上开发，进而消灭团队中数据库环境不一致，部分人员一直在错误的版本上开发的现象。

#### 版本控制和软件发布流程

版本控制和软件发布也是息息相关的。正确的一个软件测试和发布的流程可以保证我们是在正确的版本下进行测试，我们发布的最终软件是测试完成的版本。而不是错误的将一个测试中的版本或者正在开发中的版本错误的发布出去。版本控制和软件发布流程请参看下图。



附版本号命名规则：

<主版本号>.<次版本号>.<修订号>-TypeLabel三段组成，中间是点号分开。

TypeLabel的意义有如下几种：

Alpha：是内部测试版,一般不向外部发布,会有很多Bug.是内部开发完成的版本，用于测试。

Beta：测试版，这个版本会一直加入新的功能。在Alpha版之后推出。

RC：Release　Candidate)候选版本。RC版不会再加入新的功能了，主要着重于除错。

GA：General Availability,正式发布的版本，在国外都是用GA来说明release版本的。

## 引用包与依赖管理

## stub与mock设计

## 命名规范

|  |  |
| --- | --- |
|  | 定义规范的目的是为了使项目的代码样式统一，使程序有良好的可读性。  <注意事项>   * 禁止使用只有大小写区别的两个命名。 * 避免使用过长的命名。 * 为了避免混乱，尽量避免使用只有细微差别的命名。 * 进行方法命名时，需要注意单词的对称性。 |

### 构建用语辞典

|  |  |
| --- | --- |
| 构建用语词典 | |
|  | 一般来说，作成DataModel时构筑用语辞典，并与开发人员共享。  Java没有命名长度的规定，但DBMS有长度限制，都使用FullName是不可能的。  所以，用语辞典需要同时定义容易联想出FullName的缩略语。 |

|  |  |
| --- | --- |
| 1. 业务系的命名应从用语辞典派生 | |
| 重要 | 编码前必须整理好用语辞典。  换句话说，没有用语辞典就开始编码是很危险的，经常预示着项目失败。 |

|  |  |
| --- | --- |
| 1. 避免使用必须移动眼球才能读完的过长命名 | |
| 一般 | 人不是机械，不推荐使用必须移动眼球才能读完的过长命名，  有些通用的规则要求不能使用记号（数字番号）、罗马字、省略形等。但我们做的大型解决方案项目，  DB项目也多，复合语构成的长名字也多，如遵照上面这些规则，反而会使可读性变差。  应该考虑开发人员的背景（母语、业务知识、开发语言等），定义合适的命名。  如果能与DataModel有意义地联系起来，使用缩略形或罗马字来命名也可以。 |

### Class名、I/F名

|  |  |
| --- | --- |
| Class名、I/F名 | |
|  | Class或I/F的命名一般用名词。  作为例外，表示附加某某能力的I/F名，使用表示能力的形容词（-able）。 |

|  |  |
| --- | --- |
| 1. Class名或I/F名，请使用各单词（先头大文字，其余小文字）连结的形式 | |
| 重要 | Class名或I/F名，请使用各单词（先头大文字，其余小文字）连结的形式。 |
| （OK） | 01: public class SystemLogWriter {...} // OK  02: public class OrderSelectEntity {...} // OK  03: public interface DbConnection {...} // OK |
| (NG) | \* 01: public class SysLogWrt {...} // NG 缩略语不容易理解  \* 02: public class system\_log\_writer {...} // NG 连结形式违反 |

|  |  |
| --- | --- |
| 1. 例外Class名最后的单词使用Exception | |
| 重要 | 例外Class名最后的单词使用Exception。 |
| （OK） | 01: public class ApplicationException extends Exception {...} |

|  |  |
| --- | --- |
| 1. 抽象Class名的先头单词使用Abstract | |
| 重要 | 抽象Class名的先头单词使用Abstract。 |
| （OK） | 01: public abstract class AbstractEntity {...} |

|  |  |
| --- | --- |
| 1. 业务系的Class应按照指定的形式命名 | |
| 重要 | 业务系的Class应按照指定的形式命名。  命名规约（例）   |  |  |  |  | | --- | --- | --- | --- | | Class的种类 | | | Class名的形式 | | Action | 画面Action | | 机能ID + “Action” | | Batch Action | | Batch机能ID + “Action” | | Form | 画面 | FormBase | 画面ID + “FormBase” | | Form | 画面ID + “Form” | | Component | 机能内共通（公开用）  系统共通 | | 任意的命名（\*1）  \*末尾不需要附加“Component”。 | | 机能内共通 | | 机能内共通（公开用） + “Internal” | | DAO | Entity DAO | | Table物理名（去掉“\_”） + “Dao” | | Entity | | | Table物理名（去掉“\_”） + “Entity” | | 排他制御的制御Class | | | Table物理名（去掉“\_”） + “ExclusiveControl” |   （\*1）关于任意的命名，应使用能表示处理内容的英语全拼，每个英语单词的首字母大写。  （例） ShowMessage |
| （OK） | 01：public class WXX01Action {...} // 画面Actiong（机能ID: WXX01）  02：public class BXX01Action {...} // Batch Actiong（Batch机能ID: BXX01）  03: public class WXX011FormBase {...} // FormBase(画面ID: WXX011)  04: public class WXX011Form {...} // Form(画面ID: WXX011) |

|  |  |
| --- | --- |
| 1. Test Class名最后的单词使用Test | |
| 重要 | JUnit系的测试Class名，应是测试对象Class名 + “Test”。 |

### 常量名

|  |  |
| --- | --- |
| 常量名 | |
|  | 常量名一般是名词。 |

|  |  |
| --- | --- |
| 1. 常量名是各大写单词用“\_”连结 | |
| 重要 | 习惯上，用全部大写的各单词，用“\_”连结起来作为常量名。  但是，Serializable的serialVersionUID作为固定用法，是个例外。 |
| （OK） | 01: private static final long serialVersionUID = 1457142034245492583L; // 例外  02: public static final int OUTPUT\_LIMIT = 1000;  03: public static final String LANG\_ZH = "中文"; |

### 方法名

|  |  |
| --- | --- |
| 方法名 | |
|  | 方法名一般是【动词 （+ 目的语）】的形式。有些习惯用法除外。 |

|  |  |
| --- | --- |
| 1. 方法名一般是【动词 （+ 目的语）】的形式。 2. 动词都是小写，目的语部分采用首字母大写，其余小写的形式。 | |
| 重要 | 方法名一般是【动词 （+ 目的语）】的形式。  目的语的对象是自己时可省略。 |
| （OK） | 01: public boolean initialize() {...} // same as initalizeMe  02: public void terminate() {...} // same as terminateMe  03: private String getSql(String sqlId) {...}  04: public List getUserEntityList() {...} |
|  | 动词都是小写，目的语部分采用首字母大写，其余小写的形式。 |
| （OK） | 01: public boolean initialize() {...}  02: private DbStatement findStatement(String sqlId) {...}  03: private String getSql(String sqlId) {...}  04: public List getUserEntityList() {...} |

|  |  |
| --- | --- |
| 1. 属性的设定和取得方法名采用JavaBeans的形式 | |
| 重要 | 访问作为属性的实例变量时，推荐用JavaBeans的属性取得设定的形式。  （1）设定方法（setter）用【set + [property-name]】  （2）取得方法（getter）用【get + [property-name]】，或者【is + [property-name]】（真伪值的场合） |
| （OK） | 01: public void setUserName(String name) {...}  02: public String getUserName() {...}  03: public void setNewFace(boolean isNew) {...}  04: public boolean isNewFace() {...} |

|  |  |
| --- | --- |
| 1. 生成Object的方法名用【create + 目的物】的形式 | |
| 重要 | 生成Object的方法名用【create + [target-type]】的形式。 |
| （OK） | 01: public UserEntity createUserEntity() {...} |

|  |  |
| --- | --- |
| 1. 进行Object等值变换的方法名用【to + 目的物】的形式 | |
| 重要 | 进行Object等值变换的方法，依照习惯使用【to + [target-type]】的形式。  另外，进行型变换处理的方法，依照一般规则使用【convertTo + [target-type]】的形式。 |
| （OK） | 01: public String toString() {...}  02: public UserEntity convertToUserEntity(UserBean from) {...} |

|  |  |
| --- | --- |
| 1. 状态等返回真伪值的方法，不要使用让人容易误解返回值的真伪的命名。 2. 处理结果返回真伪值的方法，使用一般的命名规则。 | |
| 重要 | 状态等返回真伪值的方法，不要使用让人容易误解返回值的真伪的命名。  例如，【is + 形容词】【can + 动词】【has + 过去分词】【三单现动词】【三单现动词 + 名词】等。 |
| （OK） | 01: public boolean isEmpty() {...}  02: public boolean canGetObject() {...}  03: public boolean hasChanged() {...}  04: public boolean contains(Object obj) {...}  05: public boolean containsKey(Object key) {...} |
|  | 处理结果返回真伪值的方法，使用一般的命名规则。 |
| （OK） | 01: public boolean validateObject() {...}  02: public boolean checkFormat(String format) {...} |

### Instance变量名/Class变量名

|  |  |
| --- | --- |
| Instance变量名/Class变量名 | |
|  | Instance变量名/Class变量名一般用名词。 |

|  |  |
| --- | --- |
| 1. 变量名用第一个单词全小写，后续单词首字母大写的形式。 | |
| 重要 | 变量名用第一个单词全小写，后续单词首字母大写的形式。 |
| （OK） | 01: private static Map binder = new TreeMap();  02: private String userName; |

|  |  |
| --- | --- |
| 1. 补充：保存真伪值的变量，不要使用让人容易误解返回值的真伪的命名。 | |
| 重要 | 保存真伪值的变量，不要使用让人容易误解返回值的真伪的命名。  例如，【is + 形容词】【can + 动词】【has + 过去分词】【三单现动词】【三单现动词 + 名词】等。 |
| （OK） | 01: private boolean isEmpty;  02: private boolean hasChanged;  03: private boolean containsKey; |

### 参数名

|  |  |
| --- | --- |
| 参数名 | |
|  | 参数名一般用名词。 |

|  |  |
| --- | --- |
| 1. 参数名作为API的式样需要使用能代表它含义的名字。 2. 参数名用第一个单词全小写，后续单词首字母大写的形式。 | |
| 重要 | Javadoc里反映了API的式样，需要使用能代表参数含义的名字。  不会产生误解的场合，也可以使用缩略语。  参数名用第一个单词全小写，后续单词首字母大写的形式。 |
| （OK） | 01: public void setUserName(String userName) {...}  02: public int updateUserEntity(UserEntity oldUser, UserEntity newUser) {...} |
|  | 属性设定方法（setter），参数名与类变量名一样，因为这种方法的代码简单，不会产生混乱。  但如果有可能导致方法的记述混乱时，参数名需要多考虑下。 |
|  | 01: private String userName;  02: public void setUserName(String userName) {  03: this.userName = userName;  04: } |

### 局部变量名

|  |  |
| --- | --- |
| 局部变量名 | |
|  | 局部变量名一般用名词。 |

|  |  |
| --- | --- |
| 1. 局部变量应使用不会产生误解的较短的名字。 2. 局部变量名用第一个单词全小写，后续单词首字母大写的形式。 | |
| 重要 | 局部变量应使用不会产生误解的较短的名字。  不会产生误解的场合，比如使用首字母连结这种省略型也行。  局部变量名用第一个单词全小写，后续单词首字母大写的形式。 |
| （OK） | 01: WXX011Component component = new WXX011Component();  02: List<UserEntity> list = component.getUserEntityList();  03: for (UserEntity user : list) {  04: String userName = user.getUserName();  05: } |

|  |  |
| --- | --- |
| 1. 特殊：循环计数器从外侧顺序使用【i，j，k】。 | |
| 重要 | 循环计数器可以使用一个字符的变量【i，j，k】。  需要多个循环计数器的场景有两类。  （1）多维数组（集合）等的嵌套循环处理。  （2）多个数组（集合）同时循环处理。 |
| （OK） | 01: for (int i = 0; i < 7; i++) {  02: for (int j = 0; j < 3; j++) {  03: String dayName = dayOfWeek[i][j];  04: }  05: }  06: for (int i = 0, j = size - 1; i < size; i++, j--) {  07: descList[j] = ascList[i];  08: } |

|  |  |
| --- | --- |
| 1. 特殊：途中结果等场合可以使用无意义的名字。 | |
| 重要 | 计算过程中的值，或者动态文字列连结过程中的中间值，也可以使用无意义的名字。  注意：最后应该赋值给按规则命名（有意义）的变量。 |
| （OK） | 01: StringBuffer sb = new StringBuffer(256); // --> StringBuilder (?)  02: sb.append("UserId is not exist. [UserId]=[");  03: if (userId != null && !userId.equals("")) {  04: sb.append(userId);  05: }  06: sb.append("]");  07: String message = sb.toString(); |

### 扩展主题

#### 日文转换为拉丁字母的规则

开发过程中，我们希望在命名上能够体现业务含义。然而，对日项目中，将业务名词准确地翻译成简短易懂的英文单词，是极其困难的事情。通常情况下，为了命名统一、避免混乱，项目组需要制定一个词汇表，并不断更新维护、要求担当严格遵守、阅读代码时也要参照该表去理解代码，效率不高、效果不好。

为了解决上述问题，制定一个简单直接的日文转拉丁字母的规则，类似与中文转拼音。

* 全角汉字

规则类似与中文转汉语拼音，采用IME输入法输入目标单词时敲击的字母。变换后，还可以针对促音和长音对字母进行简化。还可以对单词进行简化，保留每个单词的第一个字。

|  |  |  |
| --- | --- | --- |
| 变换前 | 变换后 | 简化后 |
| 法人 | houjin | hojin |
| 出向 | shukkou | shuko |
| 社会保険（社保） | shakaiHoken | shaho |
| 年末調整（年調） | nenmatuchousei | nencho |

对于同一个假名，有可能对应多种输入方法/字母，为了避免混乱，统一如下表。

|  |  |
| --- | --- |
| 变换前 | 变换后 |
| し | shi |
| しゃ | sha |
| しゅ | shu |
| しょ | sho |
| じ | ji |
| じゃ | ja |
| じゅ | ju |
| じょ | jo |
| ち | chi |
| ちゃ | cha |
| ちゅ | chu |
| ちょ | cho |
| つ | tsu |
| ふ | fu |

对于一部分日语词汇可以译成简单的英文单词且不会造成理解困难的，使用英文单词。当然，为了避免混乱，建议项目组对这类单词进行管理。

|  |  |
| --- | --- |
| 变换前 | 变换后 |
| 利用者 | user |
| 名前 | name |

* 片假名

对于片假名，直接使用对应的英文单词即可。

|  |  |
| --- | --- |
| 变换前 | 变换后 |
| コード | code |
| ファイル | file |

## 测试数据组织

## 测试环境组织

# 模块集成(Integration)

## 集成策略

## 测试数据组织

## 测试环境组织

## 版本发布策略

# 系统测试(Verification)

## 测试策略

## 测试数据组织

## 测试环境组织

## 并发测试

# 移植项目专题

## 移植策略与原则

## 原系统分析

## 新系统架构设计

## 移植工程设计

## 集成策略

## 对比测试策略

## 工具运用