物联网学院

高频问题集

(2018/9月考版)

目录

[**一、 Dapper框架是什么？** **1**](#_Toc10543_WPSOffice_Level1)

[1. 通常解法 1](#_Toc24447_WPSOffice_Level2)

[2. 通用大牛级解法 1](#_Toc23780_WPSOffice_Level2)

[3. 解法对比及优缺点 2](#_Toc19683_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考 2](#_Toc5070_WPSOffice_Level2)

[5. 项目中体现经验的点 3](#_Toc19461_WPSOffice_Level2)

[6. 论坛参考 3](#_Toc18287_WPSOffice_Level2)

[**二、 Socket是什么** **3**](#_Toc24447_WPSOffice_Level1)

[1. 通常解法 4](#_Toc18487_WPSOffice_Level2)

[2. 通用大牛级解法 4](#_Toc3901_WPSOffice_Level2)

[3. 解法对比及优缺点 4](#_Toc14015_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考 4](#_Toc107_WPSOffice_Level2)

[5. 项目中体现经验的点 5](#_Toc257_WPSOffice_Level2)

[6. 论坛参考 5](#_Toc19422_WPSOffice_Level2)

[**三、什么是SQL注入？** **5**](#_Toc23780_WPSOffice_Level1)

[1. 通常解法 6](#_Toc5399_WPSOffice_Level2)

[2. 通用大牛级解法 6](#_Toc19694_WPSOffice_Level2)

[3. 解法对比及优缺点 6](#_Toc30741_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考 6](#_Toc5901_WPSOffice_Level2)

[5. 项目中体现经验的点 7](#_Toc20365_WPSOffice_Level2)

[6. 论坛参考 7](#_Toc16841_WPSOffice_Level2)

[**四、UML概述？** **8**](#_Toc19683_WPSOffice_Level1)

[1. 通常解法 8](#_Toc17420_WPSOffice_Level2)

[2. 通用大牛级解法 8](#_Toc28106_WPSOffice_Level2)

[3. 解法对比及优缺点 8](#_Toc27818_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考 9](#_Toc4784_WPSOffice_Level2)

[5. 项目中体现经验的点 10](#_Toc13277_WPSOffice_Level2)

[6. 论坛参考 10](#_Toc25861_WPSOffice_Level2)

[**五、WebForm与MVC区别和生命周期** **10**](#_Toc5070_WPSOffice_Level1)

[1. 通常解法 11](#_Toc26677_WPSOffice_Level2)

[2. 通用大牛级解法 11](#_Toc23708_WPSOffice_Level2)

[3. 解法对比及优缺点 12](#_Toc32686_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考 12](#_Toc26393_WPSOffice_Level2)

[5. 项目中体现经验的点 14](#_Toc2730_WPSOffice_Level2)

[6. 论坛参考 15](#_Toc28526_WPSOffice_Level2)

[**六、促销与秒杀解决方案** **15**](#_Toc19461_WPSOffice_Level1)

[1、通常解法 15](#_Toc31721_WPSOffice_Level2)

[2、通用大牛级解法 15](#_Toc27286_WPSOffice_Level2)

[3、解法对比及优缺点 16](#_Toc21736_WPSOffice_Level2)

[1. 浏览器端，最上层，会执行到一些JS代码 16](#_Toc23253_WPSOffice_Level2)

[2. 站点层，这一层会访问后端数据，拼html页面返回给浏览器 16](#_Toc657_WPSOffice_Level2)

[3. 服务层，向上游屏蔽底层数据细节 16](#_Toc1603_WPSOffice_Level2)

[4. 数据层，最终的库存是存在这里的，mysql是一个典型 16](#_Toc21546_WPSOffice_Level2)

[4、延伸及扩展问题回答参考 16](#_Toc17009_WPSOffice_Level2)

[5、项目中体现经验的点 19](#_Toc1751_WPSOffice_Level2)

[6、论坛参考 19](#_Toc1635_WPSOffice_Level2)

[**七、购物车解决方案** **19**](#_Toc18287_WPSOffice_Level1)

[1、 通用解法 20](#_Toc29240_WPSOffice_Level2)

[2、大牛解法 20](#_Toc15673_WPSOffice_Level2)

[3、解法对比及优缺点 21](#_Toc1768_WPSOffice_Level2)

[4、延伸及扩展问题回答参考 21](#_Toc26424_WPSOffice_Level2)

[5、项目中体现经验的点 22](#_Toc5599_WPSOffice_Level2)

[6、论坛参考 22](#_Toc17434_WPSOffice_Level2)

[**八、静态类与成员有哪些特点** **22**](#_Toc18487_WPSOffice_Level1)

[1、 通常解法 22](#_Toc20490_WPSOffice_Level2)

[2、通用大牛级解法 23](#_Toc22726_WPSOffice_Level2)

[3、解法对比及优缺点 23](#_Toc10142_WPSOffice_Level2)

[4、 延伸及扩展问题回答参考 23](#_Toc29664_WPSOffice_Level2)

[5、项目中体现经验的点 24](#_Toc18663_WPSOffice_Level2)

[6、论坛参考 25](#_Toc12249_WPSOffice_Level2)

[**九、六大设计原则都是什么** **25**](#_Toc3901_WPSOffice_Level1)

[1、 通常解法 25](#_Toc20130_WPSOffice_Level2)

[2、 大牛级解法 26](#_Toc9818_WPSOffice_Level2)

[3、解法对比及优缺点 66](#_Toc13864_WPSOffice_Level2)

[3、 延伸及扩展问题回答参考 67](#_Toc26697_WPSOffice_Level2)

[5、项目中体现经验的点 67](#_Toc31221_WPSOffice_Level2)

[6、论坛参考 67](#_Toc9913_WPSOffice_Level2)

[**十．数据库优化与注意事项** **67**](#_Toc14015_WPSOffice_Level1)

[1、 通用解法 68](#_Toc11590_WPSOffice_Level2)

[2、大牛解法  68](#_Toc7293_WPSOffice_Level2)

[3、解法对比及优缺点 68](#_Toc2546_WPSOffice_Level2)

[3 基本表扩展设计  72](#_Toc1338_WPSOffice_Level2)

[4、延伸及扩展问题回答参考 75](#_Toc10653_WPSOffice_Level2)

[5、项目中体现经验的点 77](#_Toc5678_WPSOffice_Level2)

# Dapper框架是什么？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 92% | 4星 | .Net | Dapper框架是什么？ | ORM框架 |

## 通常解法

相同点：

Dapper是.NET下一个Micro的ORM，它和Entity Framework或NHibnate不同，属于轻量级的，并且是半自动的。也就是说实体类都要自己写。它没有复杂的配置文件，一个单文件就可以了。Dapper相当于Java里的Mybatis。

不同点：

使用Dapper可以自动进行对象映射！

轻量级，单文件。  
代码就一个SqlMapper.cs文件，编译后就40K的一个很小的Dll。

## 通用大牛级解法

相同点：

支持多数据库

Dapper原理通过Emit反射IDataReader的序列队列，来快速的得到和产生对象。性能实在高高

不同点：

Dapper语法简单，如果你喜欢写原始的sql，你一定喜欢Dapper。同时团队人员也很容易上手；

Dapper 速度快，速度接近ADO.NET访问数据库的效率；

## 解法对比及优缺点

常见解法，轻量级，高性能。但它"只支持"原生SQL读写数据库，使用起来还是不太方便。很多常用的情景其实可以封装一下不用写SQL的，像EF一样，直接Add一个Entity虽然现在Dapper已经有了这样一个封装，但目前来看实在过于粗糙

通用解法，Dapper是一个轻量级的ORM框架，它是通过扩展IDbConnection接口来实现一系列的功能的。相比EF、NHibernate，它的功能较为简单。

Dapper在执行查询语句时会缓存SQL语句的相关信息，这样就保证了Dapper拥有较高的性能(原文：Dapper caches information about every query it runs, this allow it to materialize objects quickly and process parameters quickly)。

## 延伸及扩展问题回答参考

## 问题：Dapper 和关系查询

解答

 DapperDesigner 类型拥有多种关系，比如一对多（与产品）、一对一 (ContactInfo) 和多对多（客户端）。我已经试验过跨这些关系执行查询，而且 Dapper 能够处理这些关系。这绝对不像使用 Include 方法或投影表述 LINQ to EF 查询那么简单。我的 TSQL 技能被推到极限，这是因为 EF 在过去几年让我变得如此懒惰

## 项目中体现经验的点

解答

表述 SQL 并填充相关对象是我让 EF 在此背景中处理的任务，所以需要更多精力来编码。但是如果你要处理的数据量很大，那么运行时性能非常重要，这当然值得努力。在我的示例数据库中拥有 30,000 个设计器。仅有几个拥有产品。我做了一些简单的基准测试，确保我所做的是同类比较。在查看测试结果前，有些关于我如何测量的重点需要大家理解。

## 论坛参考

[http://blog.jobbole.com/101261/](http://www.cnblogs.com/ronli/archive/2011/10/26/2224654.html)

https://www.jianshu.com/p/b6cd7778567e

# Socket是什么

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 5星 | .Net | Socket是什么？ | C#技能点 |

## 通常解法

在计算机通信领域，socket 被翻译为“套接字”，它是计算机之间进行通信的一种约定或一种方式。通过 socket 这种约定，一台计算机可以接收其他计算机的数据，也可以向其他计算机发送数据。

## 通用大牛级解法

socket 的典型应用就是 Web 服务器和浏览器：浏览器获取用户输入的URL，向服务器发起请求，服务器分析接收到的URL，将对应的网页内容返回给浏览器，浏览器再经过解析和渲染，就将文字、图片、视频等元素呈现给用户。

## 解法对比及优缺点

一般解法：TCP/IP基本概念

通用解法：TCP/IP、Http、Socket的区别

## 延伸及扩展问题回答参考

Webservice与socket的区别?

两者区别:

Socket 是基于TCP/ip的传输层协议   
Webservice是基于http协议传输数据，http是基于tcp的应用层协议   
Webservice采用了基于http的soap协议传输数据。   
Socket接口通过流传输，不支持面向对象。   
Webservice 接口支持面向对象，最终webservice将对象进行序列化后通过流传输。

结论：   
Webservice采用soap协议进行通信，底层基于socket通信，webservice不需专门针对数据流的发送和接收进行处理，是一种跨平台的面向对象远程调用技术。

Socket和webservice：应用场景

Socket适用于传输大数量的数据，缺点：接口传输的数据需要手动解析，socket通信的接口协议需要自定义，比如：自定义一种字符串拼接的格式，比如自定义的xml数据，自定义麻烦之处在接口调用方和接口服务端需要互相讨论确定接口的协议内容，不方便，好处：适用于传输大数量的数据。

Webservcie，由于是遵循标准的soap协议，soap 协议的内容格式固定，soap协议传递的内容是xml数据，由于webservice是基于http的，所以简单理解为soap=http+xml，适用于没有性能要求情况下且数据传输量小，推荐在公开接口上使用webservice，因为soap协议的标准的。

## 项目中体现经验的点

双工通信优选Socket 技术实现。

## 论坛参考

<https://blog.csdn.net/cnctcom/article/details/53332808>

<https://blog.csdn.net/heyeqingquan/article/details/78004027>

# 三、什么是SQL注入？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 96% | 4星 | .Net | 什么是SQL注入？ | SQL技能点 |

## 通常解法

[SQL](https://baike.baidu.com/item/SQL)注入攻击是黑客对数据库进行攻击的常用手段之一

## 通用大牛级解法

随着B/S模式应用开发的发展，使用这种模式编写应用程序的程序员也越来越多。但是由于程序员的水平及经验也参差不齐，相当大一部分程序员在编写代码的时候，没有对用户输入数据的合法性进行判断，使应用程序存在安全隐患。用户可以提交一段数据库查询代码，根据程序返回的结果，获得某些他想得知的数据，这就是所谓的SQL Injection，即SQL注入。

## 解法对比及优缺点

一般解法：SQL注入攻击会导致的[数据库安全](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E5%AE%89%E5%85%A8/9983370)风险包括：[刷库](https://baike.baidu.com/item/%E5%88%B7%E5%BA%93/8990000)、[拖库](https://baike.baidu.com/item/%E6%8B%96%E5%BA%93/4633384)、[撞库](https://baike.baidu.com/item/%E6%92%9E%E5%BA%93/16480882)

通用解法：SQL注入攻击属于[数据库安全](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E5%AE%89%E5%85%A8/9983370)攻击手段之一，可以通过[数据库安全](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E5%AE%89%E5%85%A8/9983370)防护技术实现有效防护，数据库安全防护技术包括：数据库漏扫、[数据库加密](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E5%8A%A0%E5%AF%86/7798419)、[数据库防火墙](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E9%98%B2%E7%81%AB%E5%A2%99/5202178)、[数据脱敏](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E8%84%B1%E6%95%8F/7914656)、[数据库安全审计系统](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E5%AE%89%E5%85%A8%E5%AE%A1%E8%AE%A1%E7%B3%BB%E7%BB%9F/8402907)。。

## 延伸及扩展问题回答参考

## 总体思路

发现SQL注入位置；

判断[后台数据库](https://baike.baidu.com/item/%E5%90%8E%E5%8F%B0%E6%95%B0%E6%8D%AE%E5%BA%93)类型；

确定XP\_CMDSHELL可执行情况

发现WEB[虚拟目录](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E7%9B%AE%E5%BD%95)

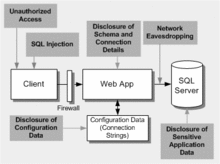
上传[ASP木马](https://baike.baidu.com/item/ASP%E6%9C%A8%E9%A9%AC)；

得到管理员权限；

## 项目中体现经验的点

### SQL注入漏洞的判断

一般来说，SQL注入一般存在于形如：HTTP://xxx.xxx.xxx/abc.asp?id=XX等带有参数SQL注入攻击的ASP[动态网页](https://baike.baidu.com/item/%E5%8A%A8%E6%80%81%E7%BD%91%E9%A1%B5)中，有时一个动态网页中可能只有一个参数，有时可能有N个参数，有时是整型参数，有时是字符串型参数，不能一概而论。总之只要是带有参数的动态网页且此网页访问了数据库，那么就有可能存在SQL注入。如果ASP程序员没有安全意识，不进行必要的字符过滤，存在SQL注入的可能性就非常大。

[](https://baike.baidu.com/pic/SQLæ³¨å¥æ»å»/4766224/0/72ccb777585a2158b051b9a9?fr=lemma%26ct=single)

## 论坛参考

<https://baike.baidu.com/item/SQL%E6%B3%A8%E5%85%A5%E6%94%BB%E5%87%BB>

# 四、UML概述？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 81% | 4星 | 软件设计 | UML概述？ | 软件设计 |

## 通常解法

UML 不是一种编程语言，但工具可用于生成各种语言的代码中使用UML图；

UML 可以用来建模非软件系统的处理流程，以及像在一个制造单元等.

## 通用大牛级解法

**UML 的目标：**

UML 的目标是定义一些通用的建模语言并对这些建模语言做出简单的说明，这样可以让建模者理解与使用。UML 也是为普通人和有兴趣的人而开发的系统，它可以是一个软件或者使用非软件，它必须是明确的。我们不将 UML 作为一个开发方法，而是随着流程做一个成功的系统。

现在我们可以明确的了解 UML 的目标就是 UML 被定义为一个简单的建模机制，帮助我们按照实际情况或者按照我们需要的样式对系统进行可视化；提供一种详细说明系统的结构或行为的方法；给出一个指导系统构造的模板；对我们所做出的决策进行文档化。

## 解法对比及优缺点

一般解法：UML 面向对象的概念

通用解法：UML 概念模型

## 延伸及扩展问题回答参考

## 问题：UML 概念模型：

对于 UML 的概念模型，我们有以下的理解：

概念模型可以被定义为模型，它是由概念和它们之间的关系组成的。

概念模型是在绘制UML图之前，它帮助了解在现实世界中的各个实体，以及他们如何互相交流。

UML 描述的实时系统，这是非常重要的一个概念模型。

掌握 UML 概念模型可以通过学习以下三大要素达到：

UML构建模块

规则连接构建模块

UML公共机制

## 问题：UML 面向对象的概念：

面向对象(Object Oriented,OO)是软件开发方法，面向对象的概念和应用已超越了程序设计和软件开发。我们可以将 UML 描述为面向对象的分析和设计的继任者。

一个对象中包含了数据和控制数据的方法，其中数据表示对象的状态，类描述的对象，他们也形成层次结构模型真实世界的系统。表示为继承层次结构，也可以以不同的方式按要求相关的类。

对象是现实世界的实体存在我们周围像抽象，封装，继承，多态的基本概念，都可以使用UML表示。因此，UML 是强大到足以代表所有的概念存在于面向对象的分析和设计。

UML 图是面向对象的概念的表示，因此，学习UML之前，详细了解面向对象的概念就变得非常重要。

以下是一些面向对象基本概念：

对象: 对象代表一个实体的基本构建块.

类: 类是对象的蓝图.

抽象化: 抽象代表现实世界中实体的行为.

封装: 封装是将数据绑定在一起，并隐藏他们外部世界的机制。

继承: 继承是从现有的机制作出新的类。

多态性: 定义的机制来以不同的形式存在.

## 项目中体现经验的点

确定一个系统中的对象.

确定它们之间的关系.

做一个设计，使用面向对象的语言可以转换为可执行文件.

## 论坛参考

https://www.w3cschool.cn/uml\_tutorial/uml\_tutorial-c1gf28pd.html

# 五、WebForm与MVC区别和生命周期

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 92% | 5星 | .Net | WebForm与MVC区别和生命周期 | WebFrom与MVC |

## 通常解法

webform生产关系是比mvc先进的，但是它不适合现在的网络设施生产力，如果要适合说不定要10年后。webform和mvc很好的印证了生产关系必须适合生产力，即使强大如微软也无法改变客观规律。

## 通用大牛级解法

MVC纵向切割了开发过程中的代码，从服务器到浏览器层层分离，层次之间耦合度很低，因为它是顺着底层的开发脉络进行封装，所以有利于开发者对整个程序过程流转的理解。但是MVC有一个非常大的缺点，这个缺点是和整个软件发展思路相背离的，那就是它无法封装、无法封装所以无法被重用。有谁看到过mvc下面的组件？有的只是一个个现成的案例，然后拿来修改。因为一个组件肯定牵涉到控制和显示，但是mvc的开发这两个层次是分离的。MVC只适合轻量级的开发，桌面开发是极少用到[mvc模式](https://www.baidu.com/s?wd=mvc%E6%A8%A1%E5%BC%8F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)的。然而[web开发](https://www.baidu.com/s?wd=web%E5%BC%80%E5%8F%91&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)恰恰就是轻量级，至今所有的[web开发](https://www.baidu.com/s?wd=web%E5%BC%80%E5%8F%91&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)都是轻量级的，因为网络硬件条件的限制，不需要也无法做到非常复杂的逻辑。这也是MVC非常非常适合[web开发](https://www.baidu.com/s?wd=web%E5%BC%80%E5%8F%91&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)的原因。  
WebForm是微软前面一套web开发的机制。它横向切割了代码，控制和显示是封装在一起的。它从开发者[思维逻辑](https://www.baidu.com/s?wd=%E6%80%9D%E7%BB%B4%E9%80%BB%E8%BE%91&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)上而不是实际情况上对代码进行封装，开发webform容易上手的原因也就在此了，但这个不利于开发者对底层程序流转机制的理解。WebForm中view和controller是放在一起的，WebForm一出现后，随之而来的是大量的组件诞生，这是[mvc模式](https://www.baidu.com/s?wd=mvc%E6%A8%A1%E5%BC%8F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)下看不到的。微软的经验之一是硬件发展很迅速。代码的封装是靠牺牲运行效率来提高开发效率，牺牲的运行效率通过提高硬件性能来解决。但微软在webform上犯了经验主义的错误，这个经验不适合网络硬件，网络硬件要考虑兼容性而且是国家的基础设施，更新的灵活性远比单机要差。大量的组件因为硬件的瓶颈无法给WebForm带来什么优势。在发展了几年webform后，微软觉得这样下去不行，等到网络硬件发展起来不知道到猴年马月了，所以就抄了一下成熟的mvc，通过Entity Framework做数据库和对象的映射，很明显，它是为了充当mvc中那个Model。通过mvc来控制和展示。

## 解法对比及优缺点

一般解法：什么是三层架构

通用解法：MVC和三层架构的区别。

## 延伸及扩展问题回答参考

什么是MVC呢？

MVC其实是软件架构的一种模式，也就是我们经常说的设计模式。其中主要包括三个模块，就是如MVC名字所显示的那样：模型（Model），视图（View），控制器（Controller）；

其中这里的模型（Model）和视图（View ）是完全区别于三层架构中的模型（Model）和视图（View）的。

1）MVC中的模型（Model）指的是数据模型，用于封装与应用程序的业务逻辑相关的数据，除此之外还可以封装数据的处理方法（相当于业务逻辑）。这是完全区别于三层架构的模型层（Model）的。

MVC中模型（Model）的特点：

①有对数据直接访问的权利，如：对数据库的访问；

②模型（Model）“不依赖”视图（View）和控制器（Controller），即模型（Model）不关心它会被如何显示或者如何被操作；

③模型（Model）中数据的变化一般会通过一种刷新机制被“公布”；

④为了实现③中的“机制”用于监视此模型的视图必须事先在此模型上注册。从而，视图可以了解在数据模型上发生的改变。

2）视图（View），这里的视图基本跟三层中的视图一样，都是为了显示数据，没有程序上的逻辑。为了实现视图上数据的刷新，视图（View）需要访问它监视的模型（Model），所以应该事先在被它监视的数据那里进行注册。

3）控制器（Controller），这个概念是在三层中不存在的概念。它主要起到不同层面的组织作用，用于控制应用程序的流程。主要处理事件并作出相应。“事件”主要包括：用户的行为和数据的改变。

以上就是关于三层架构和MVC在概念上的区别。

二，WebForm网站和MVC网站运行机制的区别

①WebForm网站的运行机制

比如说我们现在要访问一个WebForm站点：[www.google.com.hk/Default.aspx](http://www.google.com.hk/Default.aspx)(仅仅是示例)。我们的浏览器和服务器都是做了哪些动作呢？

1）首先浏览器会向目的服务器发送请求报文。

配置过IIS的都知道，网站挂载在服务器上，我们是通过访问虚拟目录的方式访问网站的。这时候目的主机的IIS接收的是访问该虚拟目录下Default.aspx文件的请求；（当然这也是一个非常复杂的过程，包括请求DNS服务器，找到目的主机IP，根据IP地址访问目的主机。复杂的网络过程就不叙述，有兴趣的自己找资料学习）；

2）服务器端的IIS软件接收到请求后，把请求交给.NET FramWork进行处理；

3）.NET FramWork会创建Default\_aspx类的对象，也就是我们所说的页面对象。（在WebFrom网站创建完，并且编译后Default.aspx会被编译成Default\_aspx类）

到现在的整个过程都还是Http请求，IIS的内部机制会去实现一个IHttphandler的接口，其中该接口实现一个ProcessRequestfang方法

## 项目中体现经验的点

新项目的搭建建议使用MVC模式

## 论坛参考

<https://blog.csdn.net/yisuowushinian/article/details/17646121>

# 六、促销与秒杀解决方案

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 98% | 4星 | .Net | 促销与秒杀解决方案 | 解决方案 |

## 1、通常解法

秒杀系统难做的原因：库存只有一份，所有人会在集中的时间读和写这些数据。例如小米手机每周二的秒杀，可能手机只有1万部，但瞬时进入的流量可能是几百几千万。又例如12306抢票，亦与秒杀类似，瞬时流量更甚。

## 2、通用大牛级解法

**主要需要解决的问题有两个：**

高并发对数据库产生的压力

竞争状态下如何解决库存的正确减少（超卖问题）

    对于第一个问题，已经很容易想到用缓存来处理抢购，避免直接操作数据库，例如使用Redis。重点在于第二个问题，常规写法：  
    查询出对应商品的库存，看是否大于0，然后执行生成订单等操作，但是在判断库存是否大于0处，如果在高并发下就会有问题，导致库存量出现负数

## 3、解法对比及优缺点

流量到了亿级别，常见站点架构如上：

1. 浏览器端，最上层，会执行到一些JS代码
2. 站点层，这一层会访问后端数据，拼html页面返回给浏览器
3. 服务层，向上游屏蔽底层数据细节
4. 数据层，最终的库存是存在这里的，mysql是一个典型

## 4、延伸及扩展问题回答参考

问题：优化方向

解答

1）将请求尽量拦截在系统上游：传统秒杀系统之所以挂，请求都压倒了后端数据层，数据读写锁冲突严重，并发高响应慢，几乎所有请求都超时，流量虽大，下单成功的有效流量甚小【一趟火车其实只有2000张票，200w个人来买，基本没有人能买成功，请求有效率为0】   
2）充分利用缓存：这是一个典型的读多写少的应用场景【一趟火车其实只有2000张票，200w个人来买，最多2000个人下单成功，其他人都是查询库存，写比例只有0.1%，读比例占99.9%】，非常适合使用缓存。

问题：优化细节

解答

1）浏览器层请求拦截

点击了“查询”按钮之后，系统那个卡呀，进度条涨的慢呀，作为用户，我会不自觉的再去点击“查询”，继续点，继续点，点点点。。。有用么？平白无故的增加了系统负载（一个用户点5次，80%的请求是这么多出来的），怎么整？

a 产品层面，用户点击“查询”或者“购票”后，按钮置灰，禁止用户重复提交请求   
b JS层面，限制用户在x秒之内只能提交一次请求

如此限流，80%流量已拦。

2）站点层请求拦截与页面缓存

浏览器层的请求拦截，只能拦住小白用户（不过这是99%的用户哟），高端的程序员根本不吃这一套，写个for循环，直接调用你后端的http请求，怎么整？

a 同一个uid，限制访问频度，做页面缓存，x秒内到达站点层的请求，均返回同一页面   
b 同一个item的查询，例如手机车次，做页面缓存，x秒内到达站点层的请求，均返回同一页面

如此限流，又有99%的流量会被拦截在站点层

3）服务层请求拦截与数据缓存  
站点层的请求拦截，只能拦住普通程序员，高级黑客，假设他控制了10w台肉鸡（并且假设买票不需要实名认证），这下uid的限制不行了吧？怎么整？

a 大哥，我是服务层，我清楚的知道小米只有1万部手机，我清楚的知道一列火车只有2000张车票，我透10w个请求去数据库有什么意义呢？对于写请求，做请求队列，每次只透有限的写请求去数据层，如果均成功再放下一批，如果库存不够则队列里的写请求全部返回“已售完”

b 对于读请求，还要我说么？cache抗，不管是memcached还是redis，单机抗个每秒10w应该都是没什么问题的

如此限流，只有非常少的写请求，和非常少的读缓存mis的请求会透到数据层去，又有99.9%的请求被拦住了

4）数据层闲庭信步  
到了数据这一层，几乎就没有什么请求了，单机也能扛得住，还是那句话，库存是有限的，小米的产能有限，透这么多请求来数据库没有意义。

5）mysql批量入库提高INSERT效率

## 5、项目中体现经验的点

## 对于秒杀系统，再次重复下两个架构优化思路： 1）尽量将请求拦截在系统上游  2）读多写少经量多使用缓存 3) redis队列缓存 + sql 批量入库

## 6、论坛参考

<https://segmentfault.com/a/1190000008888926>

**七、购物车解决方案（项目经理：李明 时长：28分41秒）**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 5星 | .Net | 购物车解决方案 | 解决方案 |

## 通用解法

从一般的商城来看，可以分为B2C与C2C，也就是单商城系统和多商城系统。单商城的系统，基本上就是全部商品生成一个订单，而多商城系统里面的购物车则是可以根据店铺来分别支付生成订单（如微店）或者全部统一支付然后根据店铺拆分订单（如有赞，淘宝等）

## 2、大牛解法

（1）根据每个店铺生成订单去支付，很好理解，例如我在店铺A，买了1，2，3这几个商品，我只需要生成一个订单号，然后去支付就可以了，后续的退款等各种处理，只需要根据该订单号进行处理即可。

（2） 那么，最后一个，购物车里面有多个店铺的商品，又需要一起支付的时候怎么办呢？假定我们使用[微信](http://lib.csdn.net/base/wechat)支付，微信支付每次下单只能使用唯一一个单号，那么我们只能把不同的店铺，例如店铺A和店铺B的所有商品，都统一放到一个订单号去微信下单支付。但是，这样子又违反了订单规则：不同的店铺存在着不同的订单业务，店铺和订单是一对多的关系，而且每个订单号必须是唯一的。怎么办？这个时候，我们可以把内部订单号和微信下单号做一个映射（也就是图所说的拆单），后续做各种处理例如退款等，就可以通过映射关系去进行处理.

## 3、解法对比及优缺点

总结一下他们之间的关系：

      （1）购物车可以存在多个店铺多个商品，可以一次性给钱购买购物车所有商品

      （2）一个订单只能对应一个店铺，一个店铺可以拥有多个订单

      （3）微信下单号只有一个，一个微信下单号可以对应多个内部订单号，一个内部订单号只能对应一个微信下单号

## 4、延伸及扩展问题回答参考

# redis解决购物车的问题

1、Redis担任读的问题，当像双11这种大量访问的情况下，Redis会不会崩溃？  
这个问题我也想过，这个我们可以考虑使用Redis的集群，这样就可以解决大部分的问题。  
2、数据库也可以做读写分离，为什么要使用Redis担任读呢，直接使用读写分离不就可以了吗？  
数据库的读写分离的确可以解决问题，但是像Redis这种非关系型数据库比较明显的优点就是数据处理效率高，读写分离和Redis的效率相比较来说，个人感觉还是使用Redis可靠。

## 5、项目中体现经验的点

在小型电商可以使用的方式比较灵活，目前来看比较优的解决方式应该是使用Redis这个非关系型数据库

## 6、论坛参考

<https://blog.csdn.net/ikownyou/article/details/53895263>

<https://blog.csdn.net/qfikh/article/details/52943068>

# 八、静态类与成员有哪些特点

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 82% | 4星 | .Net | 静态类与成员有哪些特点 | C# |

## 通常解法

[静态](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/static)类基本上与非静态类相同，但存在一个差异：静态类无法实例化。 换句话说，无法使用 [new](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/new) 关键字创建类类型的变量。 由于不存在任何实例变量，因此可以使用类名本身访问静态类的成员。

## 2、通用大牛级解法

静态类可以用作只对输入参数进行操作并且不必获取或设置任何内部实例字段的方法集的方便容器。 例如，在 .NET Framework 类库中，静态 [System.Math](https://docs.microsoft.com/zh-cn/dotnet/api/system.math) 类包含执行数学运算，而无需存储或检索对 [Math](https://docs.microsoft.com/zh-cn/dotnet/api/system.math) 类特定实例唯一的数据的方法。 即，通过指定类名和方法名称来应用类的成员

## 3、解法对比及优缺点

与所有类类型的情况一样，静态类的类型信息在引用该类的程序加载时，由 .NET Framework 公共语言运行时 (CLR) 加载。 程序无法确切指定类加载的时间。 但是，可保证进行加载，以及在程序中首次引用类之前初始化其字段并调用其静态构造函数。 静态构造函数只调用一次，在程序所驻留的应用程序域的生存期内，静态类会保留在内存中。

## 延伸及扩展问题回答参考

* 1. 静态类与非静态类基本相同，但存在一个区别：静态类不能实例化。 也就是说，不能使用 new 关键字创建静态类类型的变量。 因为没有实例变量，所以要使用类名本身访问静态类的成员。
  2. 对于只对输入参数进行运算而不获取或设置任何内部实例字段的方法集，静态类可以方便地用作这些方法集的容器。

## 5、项目中体现经验的点

以下列表提供静态类的主要功能：

只包含静态成员。

无法进行实例化。

会进行密封。

不能包含[实例构造函数](https://docs.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/instance-constructors)。

因此，创建静态类基本上与创建只包含静态成员和私有构造函数的类相同。 私有构造函数可防止类进行实例化。 使用静态类的优点是编译器可以进行检查，以确保不会意外地添加任何实例成员。 编译器可保证无法创建此类的实例。

静态类会进行密封，因此不能继承。 它们不能继承自任何类（除了 [Object](https://docs.microsoft.com/zh-cn/dotnet/api/system.object)）。 静态类不能包含实例构造函数；但是，它们可以包含静态构造函数。 如果类包含需要进行重要初始化的静态成员，则非静态类还应定义静态构造函数。 有关详细信息，请参阅[静态构造函数](https://docs.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/static-constructors)。

。

## 6、论坛参考

<https://docs.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members>

# 九、六大设计原则都是什么

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 86% | 5星 | 设计模式 | 六大设计原则都是什么 | 设计模式 |

## 通常解法

[设计模式六大原则（1）：单一职责原则](http://www.uml.org.cn/sjms/201211023.asp#1)

[设计模式六大原则（2）：里氏替换原则](http://www.uml.org.cn/sjms/201211023.asp#2)

[设计模式六大原则（3）：依赖倒置原则](http://www.uml.org.cn/sjms/201211023.asp#3)

[设计模式六大原则（4）：接口隔离原则](http://www.uml.org.cn/sjms/201211023.asp#4)

[设计模式六大原则（5）：迪米特法则](http://www.uml.org.cn/sjms/201211023.asp#5)

[设计模式六大原则（6）：开闭原则](http://www.uml.org.cn/sjms/201211023.asp#6)

## 大牛级解法

设计模式六大原则（1）：单一职责原则

定义：不要存在多于一个导致类变更的原因。通俗的说，即一个类只负责一项职责。   
问题由来：类T负责两个不同的职责：职责P1，职责P2。当由于职责P1需求发生改变而需要修改类T时，有可能会导致原本运行正常的职责P2功能发生故障。

解决方案：遵循单一职责原则。分别建立两个类T1、T2，使T1完成职责P1功能，T2完成职责P2功能。这样，当修改类T1时，不会使职责P2发生故障风险；同理，当修改T2时，也不会使职责P1发生故障风险。

说到单一职责原则，很多人都会不屑一顾。因为它太简单了。稍有经验的程序员即使从来没有读过设计模式、从来没有听说过单一职责原则，在设计软件时也会自觉的遵守这一重要原则，因为这是常识。在软件编程中，谁也不希望因为修改了一个功能导致其他的功能发生故障。而避免出现这一问题的方法便是遵循单一职责原则。虽然单一职责原则如此简单，并且被认为是常识，但是即便是经验丰富的程序员写出的程序，也会有违背这一原则的代码存在。为什么会出现这种现象呢？因为有职责扩散。所谓职责扩散，就是因为某种原因，职责P被分化为粒度更细的职责P1和P2。

比如：类T只负责一个职责P，这样设计是符合单一职责原则的。后来由于某种原因，也许是需求变更了，也许是程序的设计者境界提高了，需要将职责P细分为粒度更细的职责P1，P2，这时如果要使程序遵循单一职责原则，需要将类T也分解为两个类T1和T2，分别负责P1、P2两个职责。但是在程序已经写好的情况下，这样做简直太费时间了。所以，简单的修改类T，用它来负责两个职责是一个比较不错的选择，虽然这样做有悖于单一职责原则。（这样做的风险在于职责扩散的不确定性，因为我们不会想到这个职责P，在未来可能会扩散为P1，P2，P3，P4……Pn。所以记住，在职责扩散到我们无法控制的程度之前，立刻对代码进行重构。）

举例说明，用一个类描述动物呼吸这个场景：

|  |
| --- |
| class Animal{  public void breathe(String animal){  System.out.println(animal+"呼吸空气");  }  }  public class Client{  public static void main(String[] args){  Animal animal = new Animal();  animal.breathe("牛");  animal.breathe("羊");  animal.breathe("猪");  }  } |

运行结果：

牛呼吸空气

羊呼吸空气

猪呼吸空气

程序上线后，发现问题了，并不是所有的动物都呼吸空气的，比如鱼就是呼吸水的。修改时如果遵循单一职责原则，需要将Animal类细分为陆生动物类Terrestrial，水生动物Aquatic，代码如下：

|  |
| --- |
| class Terrestrial{  public void breathe(String animal){  System.out.println(animal+"呼吸空气");  }  }  class Aquatic{  public void breathe(String animal){  System.out.println(animal+"呼吸水");  }  }  public class Client{  public static void main(String[] args){  Terrestrial terrestrial = new Terrestrial();  terrestrial.breathe("牛");  terrestrial.breathe("羊");  terrestrial.breathe("猪");    Aquatic aquatic = new Aquatic();  aquatic.breathe("鱼");  }  } |

运行结果：

牛呼吸空气

羊呼吸空气

猪呼吸空气

鱼呼吸水

我们会发现如果这样修改花销是很大的，除了将原来的类分解之外，还需要修改客户端。而直接修改类Animal来达成目的虽然违背了单一职责原则，但花销却小的多，代码如下：

|  |
| --- |
| class Animal{  public void breathe(String animal){  if("鱼".equals(animal)){  System.out.println(animal+"呼吸水");  }else{  System.out.println(animal+"呼吸空气");  }  }  }  public class Client{  public static void main(String[] args){  Animal animal = new Animal();  animal.breathe("牛");  animal.breathe("羊");  animal.breathe("猪");  animal.breathe("鱼");  }  } |

可以看到，这种修改方式要简单的多。但是却存在着隐患：有一天需要将鱼分为呼吸淡水的鱼和呼吸海水的鱼，则又需要修改Animal类的breathe方法，而对原有代码的修改会对调用“猪”“牛”“羊”等相关功能带来风险，也许某一天你会发现程序运行的结果变为“牛呼吸水”了。这种修改方式直接在代码级别上违背了单一职责原则，虽然修改起来最简单，但隐患却是最大的。还有一种修改方式：

|  |
| --- |
| class Animal{  public void breathe(String animal){  System.out.println(animal+"呼吸空气");  }  public void breathe2(String animal){  System.out.println(animal+"呼吸水");  }  }  public class Client{  public static void main(String[] args){  Animal animal = new Animal();  animal.breathe("牛");  animal.breathe("羊");  animal.breathe("猪");  animal.breathe2("鱼");  }  } |

可以看到，这种修改方式没有改动原来的方法，而是在类中新加了一个方法，这样虽然也违背了单一职责原则，但在方法级别上却是符合单一职责原则的，因为它并没有动原来方法的代码。这三种方式各有优缺点，那么在实际编程中，采用哪一中呢？其实这真的比较难说，需要根据实际情况来确定。我的原则是：只有逻辑足够简单，才可以在代码级别上违反单一职责原则；只有类中方法数量足够少，才可以在方法级别上违反单一职责原则；

例如本文所举的这个例子，它太简单了，它只有一个方法，所以，无论是在代码级别上违反单一职责原则，还是在方法级别上违反，都不会造成太大的影响。实际应用中的类都要复杂的多，一旦发生职责扩散而需要修改类时，除非这个类本身非常简单，否则还是遵循单一职责原则的好。

遵循单一职责原的优点有：

* 可以降低类的复杂度，一个类只负责一项职责，其逻辑肯定要比负责多项职责简单的多；
* 提高类的可读性，提高系统的可维护性；
* 变更引起的风险降低，变更是必然的，如果单一职责原则遵守的好，当修改一个功能时，可以显著降低对其他功能的影响。

需要说明的一点是单一职责原则不只是面向对象编程思想所特有的，只要是模块化的程序设计，都适用单一职责原则。

设计模式六大原则（2）：里氏替换原则

肯定有不少人跟我刚看到这项原则的时候一样，对这个原则的名字充满疑惑。其实原因就是这项原则最早是在1988年，由麻省理工学院的一位姓里的女士（Barbara Liskov）提出来的。

定义1：如果对每一个类型为 T1的对象 o1，都有类型为 T2 的对象o2，使得以 T1定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型。

定义2：所有引用基类的地方必须能透明地使用其子类的对象。

问题由来：有一功能P1，由类A完成。现需要将功能P1进行扩展，扩展后的功能为P，其中P由原有功能P1与新功能P2组成。新功能P由类A的子类B来完成，则子类B在完成新功能P2的同时，有可能会导致原有功能P1发生故障。

解决方案：当使用继承时，遵循里氏替换原则。类B继承类A时，除添加新的方法完成新增功能P2外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法。

继承包含这样一层含义：父类中凡是已经实现好的方法（相对于抽象方法而言），实际上是在设定一系列的规范和契约，虽然它不强制要求所有的子类必须遵从这些契约，但是如果子类对这些非抽象方法任意修改，就会对整个继承体系造成破坏。而里氏替换原则就是表达了这一层含义。

继承作为面向对象三大特性之一，在给程序设计带来巨大便利的同时，也带来了弊端。比如使用继承会给程序带来侵入性，程序的可移植性降低，增加了对象间的耦合性，如果一个类被其他的类所继承，则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，所有涉及到子类的功能都有可能会产生故障。

举例说明继承的风险，我们需要完成一个两数相减的功能，由类A来负责。

|  |
| --- |
| class A{  public int func1(int a, int b){  return a-b;  }  }  public class Client{  public static void main(String[] args){  A a = new A();  System.out.println("100-50="+a.func1(100, 50));  System.out.println("100-80="+a.func1(100, 80));  }  } |

运行结果：

100-50=50

100-80=20

后来，我们需要增加一个新的功能：完成两数相加，然后再与100求和，由类B来负责。即类B需要完成两个功能：

* 两数相减。
* 两数相加，然后再加100。

由于类A已经实现了第一个功能，所以类B继承类A后，只需要再完成第二个功能就可以了，代码如下：

|  |
| --- |
| class B extends A{  public int func1(int a, int b){  return a+b;  }    public int func2(int a, int b){  return func1(a,b)+100;  }  }  public class Client{  public static void main(String[] args){  B b = new B();  System.out.println("100-50="+b.func1(100, 50));  System.out.println("100-80="+b.func1(100, 80));  System.out.println("100+20+100="+b.func2(100, 20));  }  } |

类B完成后，运行结果：

100-50=150

100-80=180

100+20+100=220

我们发现原本运行正常的相减功能发生了错误。原因就是类B在给方法起名时无意中重写了父类的方法，造成所有运行相减功能的代码全部调用了类B重写后的方法，造成原本运行正常的功能出现了错误。在本例中，引用基类A完成的功能，换成子类B之后，发生了异常。在实际编程中，我们常常会通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的几率非常大。如果非要重写父类的方法，比较通用的做法是：原来的父类和子类都继承一个更通俗的基类，原有的继承关系去掉，采用依赖、聚合，组合等关系代替。

里氏替换原则通俗的来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。它包含以下4层含义：

* 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。
* 子类中可以增加自己特有的方法。
* 当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。
* 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

看上去很不可思议，因为我们会发现在自己编程中常常会违反里氏替换原则，程序照样跑的好好的。所以大家都会产生这样的疑问，假如我非要不遵循里氏替换原则会有什么后果？

后果就是：你写的代码出问题的几率将会大大增加。

设计模式六大原则（3）：依赖倒置原则

定义：高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

问题由来：类A直接依赖类B，假如要将类A改为依赖类C，则必须通过修改类A的代码来达成。这种场景下，类A一般是高层模块，负责复杂的业务逻辑；类B和类C是低层模块，负责基本的原子操作；假如修改类A，会给程序带来不必要的风险。

解决方案：将类A修改为依赖接口I，类B和类C各自实现接口I，类A通过接口I间接与类B或者类C发生联系，则会大大降低修改类A的几率。

依赖倒置原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。在java中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

依赖倒置原则的核心思想是面向接口编程，我们依旧用一个例子来说明面向接口编程比相对于面向实现编程好在什么地方。场景是这样的，母亲给孩子讲故事，只要给她一本书，她就可以照着书给孩子讲故事了。代码如下：

|  |
| --- |
| class Book{  public String getContent(){  return "很久很久以前有一个阿拉伯的故事……";  }  }  class Mother{  public void narrate(Book book){  System.out.println("妈妈开始讲故事");  System.out.println(book.getContent());  }  }  public class Client{  public static void main(String[] args){  Mother mother = new Mother();  mother.narrate(new Book());  }  } |

运行结果：

妈妈开始讲故事

很久很久以前有一个阿拉伯的故事……

运行良好，假如有一天，需求变成这样：不是给书而是给一份报纸，让这位母亲讲一下报纸上的故事，报纸的代码如下：

|  |
| --- |
| class Newspaper{  public String getContent(){  return "林书豪38+7领导尼克斯击败湖人……";  }  } |

这位母亲却办不到，因为她居然不会读报纸上的故事，这太荒唐了，只是将书换成报纸，居然必须要修改Mother才能读。假如以后需求换成杂志呢？换成网页呢？还要不断地修改Mother，这显然不是好的设计。原因就是Mother与Book之间的耦合性太高了，必须降低他们之间的耦合度才行。

我们引入一个抽象的接口IReader。读物，只要是带字的都属于读物：

|  |
| --- |
| interface IReader{  public String getContent();  } |

Mother类与接口IReader发生依赖关系，而Book和Newspaper都属于读物的范畴，他们各自都去实现IReader接口，这样就符合依赖倒置原则了，代码修改为：

|  |
| --- |
| class Newspaper implements IReader {  public String getContent(){  return "林书豪17+9助尼克斯击败老鹰……";  }  }  class Book implements IReader{  public String getContent(){  return "很久很久以前有一个阿拉伯的故事……";  }  }  class Mother{  public void narrate(IReader reader){  System.out.println("妈妈开始讲故事");  System.out.println(reader.getContent());  }  }  public class Client{  public static void main(String[] args){  Mother mother = new Mother();  mother.narrate(new Book());  mother.narrate(new Newspaper());  }  } |

运行结果：

妈妈开始讲故事

很久很久以前有一个阿拉伯的故事……

妈妈开始讲故事

林书豪17+9助尼克斯击败老鹰……

这样修改后，无论以后怎样扩展Client类，都不需要再修改Mother类了。这只是一个简单的例子，实际情况中，代表高层模块的Mother类将负责完成主要的业务逻辑，一旦需要对它进行修改，引入错误的风险极大。所以遵循依赖倒置原则可以降低类之间的耦合性，提高系统的稳定性，降低修改程序造成的风险。

采用依赖倒置原则给多人并行开发带来了极大的便利，比如上例中，原本Mother类与Book类直接耦合时，Mother类必须等Book类编码完成后才可以进行编码，因为Mother类依赖于Book类。修改后的程序则可以同时开工，互不影响，因为Mother与Book类一点关系也没有。参与协作开发的人越多、项目越庞大，采用依赖导致原则的意义就越重大。现在很流行的TDD开发模式就是依赖倒置原则最成功的应用。

传递依赖关系有三种方式，以上的例子中使用的方法是接口传递，另外还有两种传递方式：构造方法传递和setter方法传递，相信用过Spring框架的，对依赖的传递方式一定不会陌生。

在实际编程中，我们一般需要做到如下3点：

* 低层模块尽量都要有抽象类或接口，或者两者都有。
* 变量的声明类型尽量是抽象类或接口。
* 使用继承时遵循里氏替换原则。

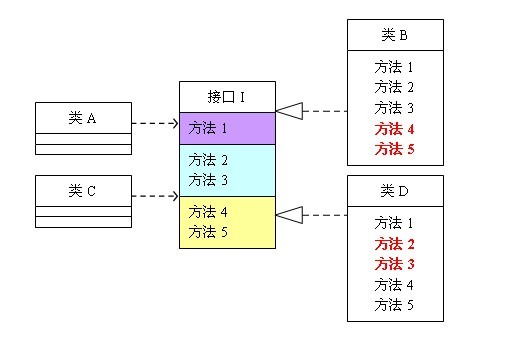
依赖倒置原则的核心就是要我们面向接口编程，理解了面向接口编程，也就理解了依赖倒置。

设计模式六大原则（4）：接口隔离原则

定义：客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。   
问题由来：类A通过接口I依赖类B，类C通过接口I依赖类D，如果接口I对于类A和类B来说不是最小接口，则类B和类D必须去实现他们不需要的方法。

解决方案：将臃肿的接口I拆分为独立的几个接口，类A和类C分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。

举例来说明接口隔离原则：

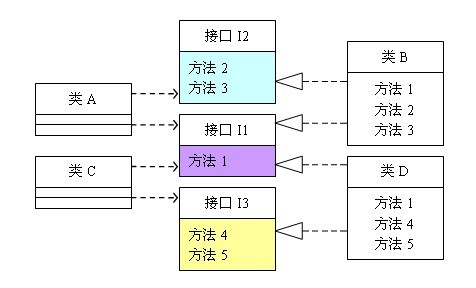


（图1 未遵循接口隔离原则的设计）

这个图的意思是：类A依赖接口I中的方法1、方法2、方法3，类B是对类A依赖的实现。类C依赖接口I中的方法1、方法4、方法5，类D是对类C依赖的实现。对于类B和类D来说，虽然他们都存在着用不到的方法（也就是图中红色字体标记的方法），但由于实现了接口I，所以也必须要实现这些用不到的方法。对类图不熟悉的可以参照程序代码来理解，代码如下：

|  |
| --- |
| interface I {  public void method1();  public void method2();  public void method3();  public void method4();  public void method5();  }  class A{  public void depend1(I i){  i.method1();  }  public void depend2(I i){  i.method2();  }  public void depend3(I i){  i.method3();  }  }  class B implements I{  public void method1() {  System.out.println("类B实现接口I的方法1");  }  public void method2() {  System.out.println("类B实现接口I的方法2");  }  public void method3() {  System.out.println("类B实现接口I的方法3");  }  //对于类B来说，method4和method5不是必需的，但是由于接口A中有这两个方法，  //所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进行实现。  public void method4() {}  public void method5() {}  }  class C{  public void depend1(I i){  i.method1();  }  public void depend2(I i){  i.method4();  }  public void depend3(I i){  i.method5();  }  }  class D implements I{  public void method1() {  System.out.println("类D实现接口I的方法1");  }  //对于类D来说，method2和method3不是必需的，但是由于接口A中有这两个方法，  //所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进行实现。  public void method2() {}  public void method3() {}  public void method4() {  System.out.println("类D实现接口I的方法4");  }  public void method5() {  System.out.println("类D实现接口I的方法5");  }  }  public class Client{  public static void main(String[] args){  A a = new A();  a.depend1(new B());  a.depend2(new B());  a.depend3(new B());    C c = new C();  c.depend1(new D());  c.depend2(new D());  c.depend3(new D());  }  } |

可以看到，如果接口过于臃肿，只要接口中出现的方法，不管对依赖于它的类有没有用处，实现类中都必须去实现这些方法，这显然不是好的设计。如果将这个设计修改为符合接口隔离原则，就必须对接口I进行拆分。在这里我们将原有的接口I拆分为三个接口，拆分后的设计如图2所示：



（图2 遵循接口隔离原则的设计）

照例贴出程序的代码，供不熟悉类图的朋友参考：

|  |
| --- |
| interface I1 {  public void method1();  }  interface I2 {  public void method2();  public void method3();  }  interface I3 {  public void method4();  public void method5();  }  class A{  public void depend1(I1 i){  i.method1();  }  public void depend2(I2 i){  i.method2();  }  public void depend3(I2 i){  i.method3();  }  }  class B implements I1, I2{  public void method1() {  System.out.println("类B实现接口I1的方法1");  }  public void method2() {  System.out.println("类B实现接口I2的方法2");  }  public void method3() {  System.out.println("类B实现接口I2的方法3");  }  }  class C{  public void depend1(I1 i){  i.method1();  }  public void depend2(I3 i){  i.method4();  }  public void depend3(I3 i){  i.method5();  }  }  class D implements I1, I3{  public void method1() {  System.out.println("类D实现接口I1的方法1");  }  public void method4() {  System.out.println("类D实现接口I3的方法4");  }  public void method5() {  System.out.println("类D实现接口I3的方法5");  }  } |

接口隔离原则的含义是：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。本文例子中，将一个庞大的接口变更为3个专用的接口所采用的就是接口隔离原则。在程序设计中，依赖几个专用的接口要比依赖一个综合的接口更灵活。接口是设计时对外部设定的“契约”，通过分散定义多个接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。

说到这里，很多人会觉的接口隔离原则跟之前的单一职责原则很相似，其实不然。其一，单一职责原则原注重的是职责；而接口隔离原则注重对接口依赖的隔离。其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口接口，主要针对抽象，针对程序整体框架的构建。

采用接口隔离原则对接口进行约束时，要注意以下几点：

* 接口尽量小，但是要有限度。对接口进行细化可以提高程序设计灵活性是不挣的事实，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度。
* 为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。
* 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。

运用接口隔离原则，一定要适度，接口设计的过大或过小都不好。设计接口的时候，只有多花些时间去思考和筹划，才能准确地实践这一原则。

设计模式六大原则（5）：迪米特法则

定义：一个对象应该对其他对象保持最少的了解。

问题由来：类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。

解决方案：尽量降低类与类之间的耦合。

自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

迪米特法则又叫最少知道原则，最早是在1987年由美国Northeastern University的Ian Holland提出。通俗的来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地的将逻辑封装在类的内部，对外除了提供的public方法，不对外泄漏任何信息。迪米特法则还有一个更简单的定义：只与直接的朋友通信。首先来解释一下什么是直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，而出现在局部变量中的类则不是直接的朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。

举一个例子：有一个集团公司，下属单位有分公司和直属部门，现在要求打印出所有下属单位的员工ID。先来看一下违反迪米特法则的设计。

|  |
| --- |
| //总公司员工  class Employee{  private String id;  public void setId(String id){  this.id = id;  }  public String getId(){  return id;  }  }  //分公司员工  class SubEmployee{  private String id;  public void setId(String id){  this.id = id;  }  public String getId(){  return id;  }  }  class SubCompanyManager{  public List<SubEmployee> getAllEmployee(){  List<SubEmployee> list = new ArrayList<SubEmployee>();  for(int i=0; i<100; i++){  SubEmployee emp = new SubEmployee();  //为分公司人员按顺序分配一个ID  emp.setId("分公司"+i);  list.add(emp);  }  return list;  }  }  class CompanyManager{  public List<Employee> getAllEmployee(){  List<Employee> list = new ArrayList<Employee>();  for(int i=0; i<30; i++){  Employee emp = new Employee();  //为总公司人员按顺序分配一个ID  emp.setId("总公司"+i);  list.add(emp);  }  return list;  }    public void printAllEmployee(SubCompanyManager sub){  List<SubEmployee> list1 = sub.getAllEmployee();  for(SubEmployee e:list1){  System.out.println(e.getId());  }  List<Employee> list2 = this.getAllEmployee();  for(Employee e:list2){  System.out.println(e.getId());  }  }  }  public class Client{  public static void main(String[] args){  CompanyManager e = new CompanyManager();  e.printAllEmployee(new SubCompanyManager());  }  } |

现在这个设计的主要问题出在CompanyManager中，根据迪米特法则，只与直接的朋友发生通信，而SubEmployee类并不是CompanyManager类的直接朋友（以局部变量出现的耦合不属于直接朋友），从逻辑上讲总公司只与他的分公司耦合就行了，与分公司的员工并没有任何联系，这样设计显然是增加了不必要的耦合。按照迪米特法则，应该避免类中出现这样非直接朋友关系的耦合。修改后的代码如下:

|  |
| --- |
| class SubCompanyManager{  public List<SubEmployee> getAllEmployee(){  List<SubEmployee> list = new ArrayList<SubEmployee>();  for(int i=0; i<100; i++){  SubEmployee emp = new SubEmployee();  //为分公司人员按顺序分配一个ID  emp.setId("分公司"+i);  list.add(emp);  }  return list;  }  public void printEmployee(){  List<SubEmployee> list = this.getAllEmployee();  for(SubEmployee e:list){  System.out.println(e.getId());  }  }  }  class CompanyManager{  public List<Employee> getAllEmployee(){  List<Employee> list = new ArrayList<Employee>();  for(int i=0; i<30; i++){  Employee emp = new Employee();  //为总公司人员按顺序分配一个ID  emp.setId("总公司"+i);  list.add(emp);  }  return list;  }    public void printAllEmployee(SubCompanyManager sub){  sub.printEmployee();  List<Employee> list2 = this.getAllEmployee();  for(Employee e:list2){  System.out.println(e.getId());  }  }  } |

修改后，为分公司增加了打印人员ID的方法，总公司直接调用来打印，从而避免了与分公司的员工发生耦合。

迪米特法则的初衷是降低类之间的耦合，由于每个类都减少了不必要的依赖，因此的确可以降低耦合关系。但是凡事都有度，虽然可以避免与非直接的类通信，但是要通信，必然会通过一个“中介”来发生联系，例如本例中，总公司就是通过分公司这个“中介”来与分公司的员工发生联系的。过分的使用迪米特原则，会产生大量这样的中介和传递类，导致系统复杂度变大。所以在采用迪米特法则时要反复权衡，既做到结构清晰，又要高内聚低耦合。

设计模式六大原则（6）：开闭原则 

定义：一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

问题由来：在软件的生命周期内，因为变化、升级和维护等原因需要对软件原有代码进行修改时，可能会给旧代码中引入错误，也可能会使我们不得不对整个功能进行重构，并且需要原有代码经过重新测试。

解决方案：当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已有的代码来实现变化。

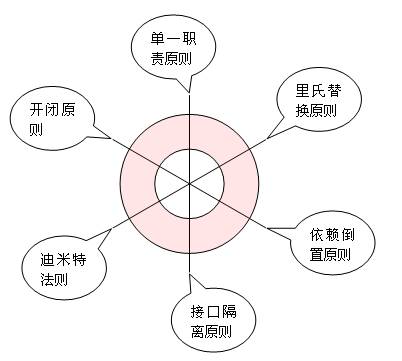
开闭原则是面向对象设计中最基础的设计原则，它指导我们如何建立稳定灵活的系统。开闭原则可能是设计模式六项原则中定义最模糊的一个了，它只告诉我们对扩展开放，对修改关闭，可是到底如何才能做到对扩展开放，对修改关闭，并没有明确的告诉我们。以前，如果有人告诉我“你进行设计的时候一定要遵守开闭原则”，我会觉的他什么都没说，但貌似又什么都说了。因为开闭原则真的太虚了。

在仔细思考以及仔细阅读很多设计模式的文章后，终于对开闭原则有了一点认识。其实，我们遵循设计模式前面5大原则，以及使用23种设计模式的目的就是遵循开闭原则。也就是说，只要我们对前面5项原则遵守的好了，设计出的软件自然是符合开闭原则的，这个开闭原则更像是前面五项原则遵守程度的“平均得分”，前面5项原则遵守的好，平均分自然就高，说明软件设计开闭原则遵守的好；如果前面5项原则遵守的不好，则说明开闭原则遵守的不好。

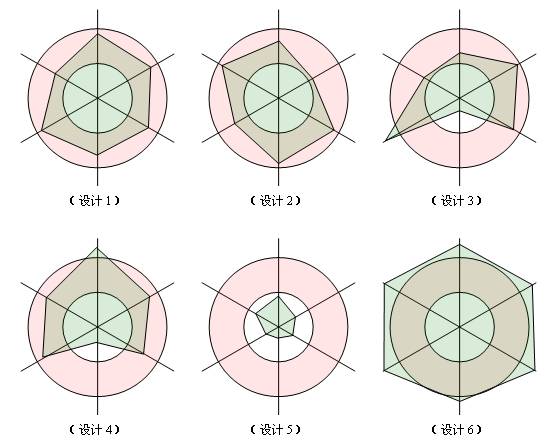
其实笔者认为，开闭原则无非就是想表达这样一层意思：用抽象构建框架，用实现扩展细节。因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。而软件中易变的细节，我们用从抽象派生的实现类来进行扩展，当软件需要发生变化时，我们只需要根据需求重新派生一个实现类来扩展就可以了。当然前提是我们的抽象要合理，要对需求的变更有前瞻性和预见性才行。

说到这里，再回想一下前面说的5项原则，恰恰是告诉我们用抽象构建框架，用实现扩展细节的注意事项而已：单一职责原则告诉我们实现类要职责单一；里氏替换原则告诉我们不要破坏继承体系；依赖倒置原则告诉我们要面向接口编程；接口隔离原则告诉我们在设计接口的时候要精简单一；迪米特法则告诉我们要降低耦合。而开闭原则是总纲，他告诉我们要对扩展开放，对修改关闭。

最后说明一下如何去遵守这六个原则。对这六个原则的遵守并不是是和否的问题，而是多和少的问题，也就是说，我们一般不会说有没有遵守，而是说遵守程度的多少。任何事都是过犹不及，设计模式的六个设计原则也是一样，制定这六个原则的目的并不是要我们刻板的遵守他们，而需要根据实际情况灵活运用。对他们的遵守程度只要在一个合理的范围内，就算是良好的设计。我们用一幅图来说明一下。



图中的每一条维度各代表一项原则，我们依据对这项原则的遵守程度在维度上画一个点，则如果对这项原则遵守的合理的话，这个点应该落在红色的同心圆内部；如果遵守的差，点将会在小圆内部；如果过度遵守，点将会落在大圆外部。一个良好的设计体现在图中，应该是六个顶点都在同心圆中的六边形。



## 3、解法对比及优缺点

设计模式的六大原则就写完了。主要参考书籍有《设计模式》《设计模式之禅》《大话设计模式》以及网上一些零散的文章，但主要内容主要还是我本人对这六个原则的感悟。写出来的目的一方面是对这六项原则系统地整理一下，一方面也与广大的网友分享，因为设计模式对编程人员来说，的确非常重要。正如有句话叫做一千个读者眼中有一千个哈姆雷特，如果大家对这六项原则的理解跟我有所不同，欢迎留言，大家共同探讨。

## 延伸及扩展问题回答参考

面向对象的关键是什么？

在于封装，封装好了才能很好的复用，达到单一职责和开放扩展、封闭更改的效果

## 5、项目中体现经验的点

在上图中，设计1、设计2属于良好的设计，他们对六项原则的遵守程度都在合理的范围内；设计3、设计4设计虽然有些不足，但也基本可以接受；设计5则严重不足，对各项原则都没有很好的遵守；而设计6则遵守过渡了，设计5和设计6都是迫切需要重构的设计。

## 6、论坛参考

<http://www.uml.org.cn/sjms/201211023.asp>

<https://segmentfault.com/a/1190000000691175>

# 十．数据库优化与注意事项

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 5星 | .Net | 数据库优化与注意事项 | 数据库设计 |

## 通用解法

数据库优化与注意事项

## 2、大牛解法

数据库优化的目标无非是避免磁盘I/O瓶颈、减少CPU利用率和减少资源竞争。为了便于读者阅读和理解，笔者参阅了Sybase、Informix和Oracle等大型数据库系统参考资料，基于多年的工程实践经验，从基本表设计、扩展设计和数据库表对象放置等角度进行讨论，着重讨论了如何避免磁盘I/O瓶颈和减少资源竞争，相信读者会一目了然

## 3、解法对比及优缺点

**基于第三范式的基本表设计**

在基于表驱动的信息管理系统（MIS）中，基本表的设计规范是第三范式（3NF）。第三范式的基本特征是非主键属性只依赖于主键属性。基于第三范式的数据库表设计具有很多优点：

1. 消除了冗余数据，节省了磁盘存储空间；
2. 有良好的数据完整性限制，即基于主外键的参照完整限制和基于主键的实体完整性限制，这使得数据容易维护，也容易移植和更新；
3. 数据的可逆性好，在做连接（Join）查询或者合并表时不遗漏、也不重复；
4. 因消除了冗余数据（冗余列），在查询（Select）时每个数据页存的数据行就多，这样就有效地减少了逻辑I/O，每个Cash存的页面就多，也减少物理I/O；
5. 对大多数事务(Transaction)而言，运行性能好；
6. 物理设计(Physical Design)的机动性较大，能满足日益增长的用户需求。

在基本表设计中，表的主键、外键、索引设计占有非常重要的地位，但系统设计人员往往只注重于满足用户要求，而没有从系统优化的高度来认识和重视它们。实际上，它们与系统的运行性能密切相关。现在从系统数据库优化角度讨论这些基本概念及其重要意义：

**主键(Primary Key)：**主键被用于复杂的SQL语句时，频繁地在数据访问中被用到。一个表只有一个主键。主键应该有固定值（不能为Null或缺省值，要有相对稳定性），不含代码信息，易访问。把常用（众所周知）的列作为主键才有意义。短主键最佳（小于25bytes），主键的长短影响索引的大小，索引的大小影响索引页的大小，从而影响磁盘I/O。主键分为自然主键和人为主键。自然主键由实体的属性构成，自然主键可以是复合性的，在形成复合主键时，主键列不能太多，复合主键使得Join\*作复杂化、也增加了外键表的大小。人为主键是，在没有合适的自然属性键、或自然属性复杂或灵敏度高时，人为形成的。人为主键一般是整型值（满足最小化要求），没有实际意义，也略微增加了表的大小；但减少了把它作为外键的表的大小。

**外键（Foreign Key）：**外键的作用是建立关系型数据库中表之间的关系（参照完整性），主键只能从独立的实体迁移到非独立的实体，成为后者的一个属性，被称为外键。

**索引(Index)：**利用索引优化系统性能是显而易见的，对所有常用于查询中的Where子句的列和所有用于排序的列创建索引，可以避免整表扫描或访问，在不改变表的物理结构的情况下，直接访问特定的数据列，这样减少数据存取时间；利用索引可以优化或排除耗时的分类\*作；把数据分散到不同的页面上，就分散了插入的数据；主键自动建立了唯一索引，因此唯一索引也能确保数据的唯一性（即实体完整性）；索引码越小，定位就越直接；新建的索引效能最好，因此定期更新索引非常必要。索引也有代价：有空间开销，建立它也要花费时间，在进行Insert、Delete和Update\*作时，也有维护代价。索引有两种：聚族索引和非聚族索引。一个表只能有一个聚族索引，可有多个非聚族索引。使用聚族索引查询数据要比使用非聚族索引快。在建索引前，应利用数据库系统函数估算索引的大小。

* **聚族索引（Clustered Index）：**聚族索引的数据页按物理有序储存，占用空间小。选择策略是，被用于Where子句的列：包括范围查询、模糊查询或高度重复的列（连续磁盘扫描）；被用于连接Join\*作的列；被用于Order by和Group by子句的列。聚族索引不利于插入\*作，另外没有必要用主键建聚族索引。
* **非聚族索引（Nonclustered Index）：**与聚族索引相比，占用空间大，而且效率低。选择策略是，被用于Where子句的列：包括范围查询、模糊查询（在没有聚族索引时）、主键或外键列、点（指针类）或小范围（返回的结果域小于整表数据的20%）查询；被用于连接Join\*作的列、主键列（范围查询）；被用于Order by和Group by子句的列；需要被覆盖的列。对只读表建多个非聚族索引有利。

索引也有其弊端，一是创建索引要耗费时间，二是索引要占有大量磁盘空间，三是增加了维护代价（在修改带索引的数据列时索引会减缓修改速度）。

那么，在哪种情况下不建索引呢？对于小表（数据小于5页）、小到中表（不直接访问单行数据或结果集不用排序）、单值域（返回值密集）、索引列值太长（大于20bitys）、容易变化的列、高度重复的列、Null值列，对没有被用于Where子语句和Join查询的列都不能建索引。另外，对主要用于数据录入的，尽可能少建索引。当然，也要防止建立无效索引，当Where语句中多于5个条件时，维护索引的开销大于索引的效益，这时，建立临时表存储有关数据更有效。

批量导入数据时的注意事项：在实际应用中，大批量的计算（如电信话单计费）用C语言程序做，这种基于主外键关系数据计算而得的批量数据（文本文件），可利用系统的自身功能函数（如Sybase的BCP命令）快速批量导入，在导入数据库表时，可先删除相应库表的索引，这有利于加快导入速度，减少导入时间。在导入后再重建索引以便优化查询。

**锁：**锁是并行处理的重要机制，能保持数据并发的一致性，即按事务进行处理；系统利用锁，保证数据完整性。因此，我们避免不了死锁，但在设计时可以充分考虑如何避免长事务，减少排它锁时间，减少在事务中与用户的交互，杜绝让用户控制事务的长短；要避免批量数据同时执行，尤其是耗时并用到相同的数据表。锁的征用：一个表同时只能有一个排它锁，一个用户用时，其它用户在等待。若用户数增加，则Server的性能下降，出现“假死”现象。如何避免死锁呢？从页级锁到行级锁，减少了锁征用；给小表增加无效记录，从页级锁到行级锁没有影响，若在同一页内竞争有影响，可选择合适的聚族索引把数据分配到不同的页面；创建冗余表；保持事务简短；同一批处理应该没有网络交互。

**查询优化规则：**在访问数据库表的数据(Access Data)时，要尽可能避免排序（Sort）、连接(Join)和相关子查询\*作。经验告诉我们，在优化查询时，必须做到：

* 尽可能少的行；
* 避免排序或为尽可能少的行排序，若要做大量数据排序，最好将相关数据放在临时表中\*作；用简单的键（列）排序，如整型或短字符串排序；
* 避免表内的相关子查询；
* 避免在Where子句中使用复杂的表达式或非起始的子字符串、用长字符串连接；
* 在Where子句中多使用“与”（And）连接，少使用“或”(Or)连接；
* 利用临时数据库。在查询多表、有多个连接、查询复杂、数据要过滤时，可以建临时表（索引）以减少I/O。但缺点是增加了空间开销。

除非每个列都有索引支持，否则在有连接的查询时分别找出两个动态索引，放在工作表中重新排序。

**3 基本表扩展设计**

基于第三范式设计的库表虽然有其优越性（见本文第一部分），然而在实际应用中有时不利于系统运行性能的优化：如需要部分数据时而要扫描整表，许多过程同时竞争同一数据，反复用相同行计算相同的结果，过程从多表获取数据时引发大量的连接\*作，当数据来源于多表时的连接\*作；这都消耗了磁盘I/O和CPU时间。  
尤其在遇到下列情形时，我们要对基本表进行扩展设计：许多过程要频繁访问一个表、子集数据访问、重复计算和冗余数据，有时用户要求一些过程优先或低的响应时间。   
如何避免这些不利因素呢？根据访问的频繁程度对相关表进行分割处理、存储冗余数据、存储衍生列、合并相关表处理，这些都是克服这些不利因素和优化系统运行的有效途径。

**3.1 分割表或储存冗余数据**分割表分为水平分割表和垂直分割表两种。分割表增加了维护数据完整性的代价。   
水平分割表：一种是当多个过程频繁访问数据表的不同行时，水平分割表，并消除新表中的冗余数据列；若个别过程要访问整个数据，则要用连接\*作，这也无妨分割表；典型案例是电信话单按月分割存放。另一种是当主要过程要重复访问部分行时，最好将被重复访问的这些行单独形成子集表（冗余储存），这在不考虑磁盘空间开销时显得十分重要；但在分割表以后，增加了维护难度，要用触发器立即更新、或存储过程或应用代码批量更新，这也会增加额外的磁盘I/O开销。   
垂直分割表（不破坏第三范式），一种是当多个过程频繁访问表的不同列时，可将表垂直分成几个表，减少磁盘I/O（每行的数据列少，每页存的数据行就多，相应占用的页就少），更新时不必考虑锁，没有冗余数据。缺点是要在插入或删除数据时要考虑数据的完整性，用存储过程维护。另一种是当主要过程反复访问部分列时，最好将这部分被频繁访问的列数据单独存为一个子集表（冗余储存），这在不考虑磁盘空间开销时显得十分重要；但这增加了重叠列的维护难度，要用触发器立即更新、或存储过程或应用代码批量更新，这也会增加额外的磁盘I/O开销。垂直分割表可以达到最大化利用Cache的目的。   
总之，为主要过程分割表的方法适用于：各个过程需要表的不联结的子集，各个过程需要表的子集，访问频率高的主要过程不需要整表。在主要的、频繁访问的主表需要表的子集而其它主要频繁访问的过程需要整表时则产生冗余子集表。   
注意，在分割表以后，要考虑重新建立索引。

**3.2 存储衍生数据**对一些要做大量重复性计算的过程而言，若重复计算过程得到的结果相同（源列数据稳定，因此计算结果也不变），或计算牵扯多行数据需额外的磁盘I/O开销，或计算复杂需要大量的CPU时间，就考虑存储计算结果（冗余储存）。现予以分类说明：   
若在一行内重复计算，就在表内增加列存储结果。但若参与计算的列被更新时，必须要用触发器更新这个新列。   
若对表按类进行重复计算，就增加新表（一般而言，存放类和结果两列就可以了）存储相关结果。但若参与计算的列被更新时，就必须要用触发器立即更新、或存储过程或应用代码批量更新这个新表。   
若对多行进行重复性计算（如排名次），就在表内增加列存储结果。但若参与计算的列被更新时，必须要用触发器或存储过程更新这个新列。

总之，存储冗余数据有利于加快访问速度；但违反了第三范式，这会增加维护数据完整性的代价，必须用触发器立即更新、或存储过程或应用代码批量更新，以维护数据的完整性。

**3.3 消除昂贵结合**对于频繁同时访问多表的一些主要过程，考虑在主表内存储冗余数据，即存储冗余列或衍生列（它不依赖于主键），但破坏了第三范式，也增加了维护难度。在源表的相关列发生变化时，必须要用触发器或存储过程更新这个冗余列。当主要过程总同时访问两个表时可以合并表，这样可以减少磁盘I/O\*作，但破坏了第三范式，也增加了维护难度。对父子表和1：1关系表合并方法不同：合并父子表后，产生冗余表；合并1：1关系表后，在表内产生冗余数据。

## 4、延伸及扩展问题回答参考

问题：redis的存储方式（键值对的存储方式）

答：现在比较常用的是键值对的存储方式，说到存储方式，需用从哈希算法说起，hash算法将对象转换为一个散列①，我们在使用的时候会将这个散列转换为一个大数，这就是hashcode，而hashmap高度依赖于hash和hashcode

hashtable默认会开辟4个存储空间，将获取到的hashcode求磨，算出下标志②。也就是说hashtable是键值对的存储方式，

我们一般会用Hashmap分表分库，hashmap的原理和hashtable相似，他们之间的区别在这里就不细说了，hashmap分表分库也是对hashcode求魔然后找下标值，但是下标值里存在数据的话，会自动生成链路（在hashmap中的是倒链③）

注释：

①：HashMap 之所以速度快，因为他使用的是散列表，根据 key 的 hashcode 值生成数组下标（通过内存地址直接查找，没有任何判断），时间复杂度完美情况下可以达到 n1（和数组相同，但是比数组用着爽多了，但是需要多出很多内存，相当于以空间换时间）

②：假若hashcode为112，hashtable为4个空间，那么这个数据存储的下标值为 112%4=0，也就是说存在了下标为0的空间

③：链表存储是不连续的，查询慢，插入块。数组是连续的，查询快，插入慢

## 项目中体现经验的点 ##

在项目中使用了Redis做多点登录，将生成的token保存在redis中，在数据查询的过程中，先从缓存读取数据，加快页面运行速度，减轻服务器压力

## 5、项目中体现经验的点

数据库对象的放置策略是均匀地把数据分布在系统的磁盘中，平衡I/O访问，避免I/O瓶颈。

1. 访问分散到不同的磁盘，即使用户数据尽可能跨越多个设备，多个I/O运转，避免I/O竞争，克服访问瓶颈；分别放置随机访问和连续访问数据。
2. 分离系统数据库I/O和应用数据库I/O。把系统审计表和临时库表放在不忙的磁盘上。
3. 把事务日志放在单独的磁盘上，减少磁盘I/O开销，这还有利于在障碍后恢复，提高了系统的安全性。
4. 把频繁访问的“活性”表放在不同的磁盘上；把频繁用的表、频繁做Join\*作的表分别放在单独的磁盘上，甚至把把频繁访问的表的字段放在不同的磁盘上，把访问分散到不同的磁盘上，避免I/O争夺；
5. 利用段分离频繁访问的表及其索引（非聚族的）、分离文本和图像数据。段的目的是平衡I/O，避免瓶颈，增加吞吐量，实现并行扫描，提高并发度，最大化磁盘的吞吐量。利用逻辑段功能，分别放置“活性”表及其非聚族索引以平衡I/O。当然最好利用系统的默认段。另外，利用段可以使备份和恢复数据更加灵活，使系统授权更加灵活。

## 6、论坛参考

<http://database.51cto.com/art/201103/247427.htm>