之前给团队编制过一份数据库设计规范，详细总结了数据库设计、使用过程中的种种细节问题，并给出了相应的解决方案。编写制定本文档的过程中，从网络上翻阅了数百份数据库设计相关的文章、帖子，应该是认真用心全面的一份关系型数据库设计相关的文档。

**一 文档说明**

**1、引言**

“文档说明”部分将给出本数据库设计规范的编写背景，并从全局介绍数据库设计过程中各环节、各模块的规范要求。

**2、编写背景**

“HEBSMJYPT”的项目告一段落后，着手开始“JCFGTGYLJRDSJ”项目。所在单位所领团队之前的数据库设计工作是按业务模块划分后交由不同的开发人员负责，没有统一规范，组合在一起的完整设计相对杂乱。数据库设计的完整性、统一性被破坏后，程序部分自然也会受到影响，进而影响整个项目的稳定性、协调性。即便是过去独立负责设计的数据库，在细节上也有诸多不合理的地方，其中一个重要的诱导因素就是不统一——设计理念、字段类型、命名方式、通用表通用字段的处理方法等等因为没有规范的约束而导致的不统一。

所以在开始“JCFGTGYLJRDSJ”项目的设计之前，着手制定出一套标准合理的数据库设计规范，至此之后的项目，数据库设计工作无论由一人负责还是多人协作，都严格依此规范进行，籍此避免再次出现早前的许多问题。

在制定本规范之前，从各种渠道梳理汇总了数百份数据库设计规范相关的文档，也回看了过去负责或参与项目的数据库结构图，把好的设计方案加以总结沉淀、把尚有问题的地方列出并给出相应的解决方法，根据以往经验和现实需要将这些种种去其糟粕取其精华，汇成一家。

**3、文档概述**

整个规范的制定可以分成两部分，第一部分是设计规范，第二部分是命名规范。命名规范也属于设计规范的模块，但因其比较重要，且涉及的内容较多，故将其剥离单独介绍。本来文档编辑之初还想写一个操作规范模块，操作规范用来约束数据库设计完成之后的工作，比如程序编写过程中SQL语句对数据的增删改查，开发过程中对表结构的增删改，维护过程中对数据库的备份迁移等等。但因为时间和个人缘故，最终决定拿掉此模块，此模块中的一些涉及点融合到了设计规范和命名规范中进行介绍。在文档的最后，也会给出此规范编写过程中的参考文献，多是网络上个人撰写的数据库设计规范相关的非正式文章。

中小型的Web项目，数据库以MySQL、SQLServer及Oralce为主，非关系型数据库不在本规范的考虑范围之内，而不同的关系型数据库设计理念大同小异，不过具体到细节上，又可以说差异巨大。鉴于当前开发面临的实际情况，本规范的制定主要针对（但不限于）MySQL，对于特别部分，会对比不同的关系型数据库具体说明，如无特别说明，则默认表示MySQL数据库。

此为1.0版本，制定基本规范，后面会持续更新，根据不同的项目、不同的数据库、不同的程序架构、不同的开发模式、不同的数据量并发访问量等实际情况，在现有规范的基础上完善修正，越是后期的版本，规范会愈加全面、愈加灵活、愈加合理、适应性愈加更广阔。

**二 设计规范**

**1、引言**

设计规范部分目前八个子模块：设计理念、基本原则、字段设置、相关注释、约束控制、索引添加、特别说明及梳理总结部分，命名规范部分单独列出一个大模块，在后面的章节介绍。本章节是文档的核心部分，详细说明数据库设计步骤、设计原理、设计过程中各环节要遵守的原则、注意事项等。

目前的设计规范并不全面，存储过程、触发器、函数、视图、事件等高级功能的设计规范都未特别说明，原因后面会讲。在后续版本的规范文档中，可根据实际情况修改扩充。

**2、设计理念**

A.设计流程

从项目启动之时，数据库设计工作就已经开始，贯穿于项目前期阶段的需求调研、分析、确认、业务梳理过程，只不过这时的设计大都停留在脑海中，正式的设计过程起始于最终的需求确认完成、业务梳理清晰之后。

就目前来看，最好的关系型数据库设计工具还是PowerDesigner（以后简称PD），我们要求正式数据库设计过程中必须使用此工具，先出CDM（Concept Data Model，概念数据模型），再根据实际的数据库类型由CDM导出PDM（Physical Data Model，物理数据模型），最后由PDM将设计成果直接导入到数据库中。同时导出相应的数据库文档，以供项目组开发人员查阅。

CDM设计过程中只做最简单必要的设计，约束、关系、主键、外键、命名规则等在转换成PDM过程中PD可自行处理的部分均交由工具自行处理。就是说数据库设计者只负责工具无法处理的少许部分，工具可完成的部分则用工具统一处理，这样设计工作会更高效省时，最终的设计成果也会更规范合理。

初版数据库设计完成之后进入项目开发阶段，如果前期的需求调研分析、业务梳理没问题，后期数据库结构发生大规模改动的情况不应该频繁出现的。但需求总是在变，意外才是唯一的法则，即便设计者在前期调研、了解、分析、设计的过程中再过谨慎，怕只能减少这种情况出现的风险，却不能完全避免。还有随着开发的深入，数据库结构发生局部变动，比如增删改些表字段等也是再正常不过的了。针对这些，都应该有相应的对策，才能以不变应万变。

第一版的数据库设计完成，由PDM生成相应的SQL脚本在数据库中执行之后，随开发深入而再进行的数据库改动分两种情况：一种是大改，比如原有的业务有变动，或因在设计时考虑不周、对需求了解不清导致设计出错，表及引用关系都要发生重大变化，刚也说了这种情况不应该出现；另一种是小改，比如有新增业务的情况要新建些表，有拿掉部分业务的情况可能会删除些表，还有更常出现的是增删改部分表的部分字段。

对于以上两种，有增加表或大规模业务变动的情况，建议是在PD中修改PDM，然后重新生成SQL脚本在数据库中执行，当然只生成新增表或发生业务变动部分的即可，而对于简单的增删改些字段或业务变动不大的情况，建议直接在数据库中对表进行修改。然后通过PD菜单中的Database——Update Model from Database……连接数据库逆向更新PDM，使PDM和最新的数据库结构保持一致，而后再生成最新的数据库文档。但要注意的是，使用此功能逆向更新PDM，只会更新修改或新增的表字段，而不更新删除的字段。就是说，如果在数据库中对一个表的字段进行了修改，或者新增了一些字段，PD会同步在PDM中的相应表中做出相应的修改或新增，但如果在数据库中删除了表中的某个字段，PD是不会删除PDM中相应表的相应字段的。为什么这般处理，自己也觉得很奇怪。

后期的数据库改动，一般发生在程序开发启动后，如果主程序可自动生成，第二种小改的情况还是比较容易处理的，重新生成下主程序，而手写的部分（非自动生成部分）一般不会受到太大影响。但如果是第一种情况，业务发生了变动，那可能意味着手写的程序（非自动生成部分）要重写。之前的文章中也有提及，一定要注意前期的需求调研了解分析系统设计，后期的问题几乎都是由前期的不慎造成，有经验的项目经理可以在前期预料到后面可能的问题而提前采取相应的预防措施。防之于未有，治之于未乱。项目想要做的出色，有太多不可测因素，但如果手底下的项目都很稳当，其能力必是值得肯定的。

这里还要提一点，后期数据库表小范围的修改一般是由开发人员发现，比如在开发过程中发现少了一些字段，或局部业务有些问题等等，应该禁止开发人员擅自直接更改数据库，所有更改无论大小必须经过数据库主设计师的审核同意，以避免可能影响到全局的更改出现。

数据库的设计工作虽然集中在项目的业务梳理清晰之后、正式开发之前，但相关细枝末节的工作却不止于此，很可能会贯穿于整个项目的起始流程。

B.三种关系

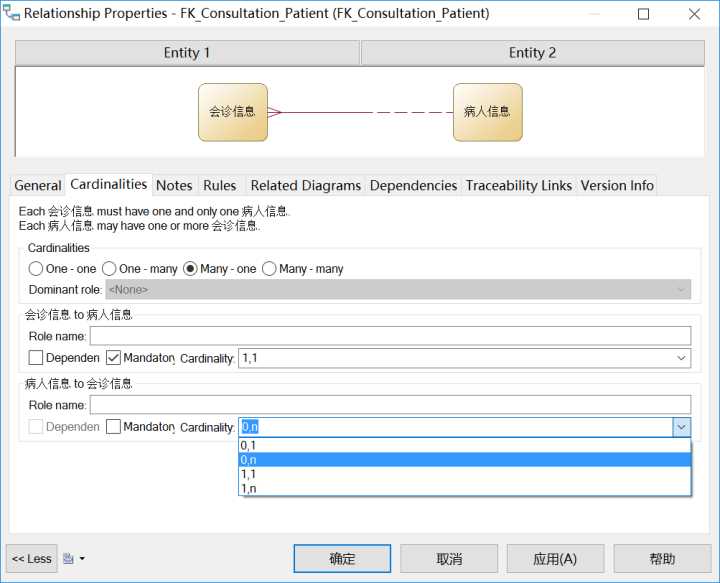
有人说数据库难以设计，其实难的并不是数据库的设计，而是业务流程的梳理。再复杂的业务，只要理得清，表现在数据库中，无外乎是表与表间的三种关系：一对一（one-to-one）、一对多（one-to-many ）以及多对多（many-to-many）。更进一步的，many-to-many实际上就是两个one-to-many。

在Java中万事万物皆对象，在关系型数据库中万事万物皆是二维表，而事物之间的联系就是表与表间的这三种关系。

后面还会多次提及，我们的设计原则是尽可能让粒度小、容忍度高，比如在“设计规范”——“字段设置”——“通用字段处理”中有关于日期时间类型设置的说明，要求日期时间类型的字段，尽可能用datetime类型，精确到时分秒，而不要用date类型。表现在这里，处理业务关系时，对于核心业务部分尚不能明确表与表关系的，能一对多就不要一对一，能多对多就不要一对多。这样开发的复杂度会增加，却消除了后面可能的修改扩展的隐患。对于非核心业务也不能明确关系的，可根据实际情况，综合考量开发实现的烦琐程度及未来的可变性再做决定。

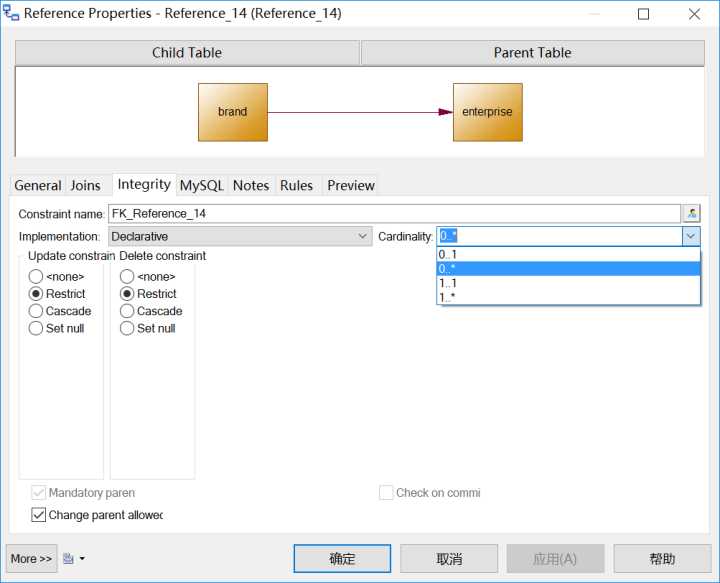
PD细化了这三种关系的表述，表现在CDM关系中Cardinalities选项卡的Cardinality选项中、表现在PDM引用中Integrity选项卡的Cardinality选项中。Cardinality，基数，在CDM的选项中表示另一方对于当前方的每个实例，可能拥有的实例的最少和最多数；在PDM的选项中表示父表中的每个实例，子表中可能拥有的实例的最少和最多数。

比如“病人”与“会诊单”两个实体之间的联系是one-to-many联系，换个方向说“会诊单”和“病人”之间的联系是many-to-one联系。而且一个会诊单必须属于一个病人，并且只能属于一个病人，不能属于零个病人，所以从“会诊单”实体至“病人”实体的基数为“1,1”；从联系的另一方向考虑，一个病人可以拥有多个会诊单，也可以没有任何会诊单，即零个会诊单，所以该方向联系的基数就为“0,n”。CDM中的表示如下图所示：



在构建CDM、选择两个实体之间的关系时，这部分会自动赋值的。

类似的，一个品牌必需且只能属于一个企业，一个企业却可以有一个或多个品牌，又或者一个也没有。在PDM的选项中表示父表为企业、子表为品牌，父表（企业）中的每个实例，子表（品牌）中可能拥有的实例的最少和最多数。如下图所示：



PD的细化在于，one-to-one关系中到底是must have one and only one还是may have at most one，one-to-many关系中到底是must have one or more还是may have one or more。many-to-many关系需要借助中间表实现，分解成两个one-to-many组合，类似的，也可被细分成严格的多对多还是模糊的多对多。可以拿用户和角色的例子说明，严格的多对多是一个用户至少有一个角色、一个角色至少被一个用户拥有，模糊的多对多是一个用户可以没有任何角色、一个角色可以不被任何用户拥有。

个人认为对one-to-many和many-to-many两种关系进行细化的意义不大，因为对于one-to-many、many-to-one，无论是否严格，都要在many方加入引用one方主键的外键；而对于many-to-many，无论是否严格，中间表都是必须的。所以对于这两种关系，无论是否为严格的，设计方案是确定的。

但对于one-to-one，严格的和非严格关系之间设计上是有区别的。比如订单和取消原因，一个订单可以有一个或零个取消原因（may have at most one ），本来如果是严格的一对一关系，取消原因是可以整合到订单表中的，而无需单建表存储。 再比如班级与班长，一个班级只有一个正班长， 一个班长只在一个班中任职，两边都是must have one and only one，那班长表完全可以省略，而直接在班级表中加入相应描述班长信息的字段即可，比如班长名称、性别等等，又或者直接有个班长的外键字段指向学生表。

C.如何设计

拿到项目后，完成早期的需求调研，在分析设计的过程中，先考虑的是要实现所需功能、需要的实体有哪些。比如要实现登录功能，那必需要有用户实体。继续扩展思路，登录往往伴随着角色划分，为此要有角色、权限相关的实体。为了记录登录请求，又要有登录日志实体。这些实体表现在数据库即为相应的表，实体确定，同步考虑实体与实体间的关系。一个用户可以有多个角色，一个角色可以被多个用户拥有，所以用户和角色是多对多关系，多对多关系意味着出现中间表。一个用户可以有多条登录日志，一条登录日志有且只能对应一个用户，所以用户和登录日志是一对多的关系，登录日志表中要有外键引用用户表。这就是数据库设计的过程。

数据库的设计往往和前端界面的设计并行，前者稍晚于后者，两者的进行伴随在业务梳理的过程中、前期需求确认之后，为确保对需求理解的准确性、设计的准确性，此过程中应该继续和需求提出者保持沟通。如果在设计数据库时不考虑前端界面设计、或者是在设计界面时不考虑数据库设计，且不说两边对业务需求的理解可能有偏差，即便完全没有，最终怕也不能完好无误的进行融合，这也是为什么觉得项目经理、产品经理、技术经理的角色一人扮演要比三人分饰更好的原因。成熟的社会体制下，人与人间的分工应该更加明确，这无可厚非，问题就出在很多公司分工提前明确了，协作体制却不完善。如此这般，做一件事情参与的人越多，不但不会节省工时，反而导致的问题越多，大大降低了工作效率。

负责数据库设计工作的人应该是最懂项目、最懂业务需求、最有设计经验的人，此人必须跟踪整个项目的设计开发过程，产品界面的设计、程序的开发都要和其沟通确认方可。项目设计开发实施过程中，很多决定，不是参与决定的人越多越好，而是由一个最懂项目的人决定最好。有时为了某个决定组织集体讨论，最终的决定往往来自于主导讨论者，而非是最懂项目者。做项目和带兵打仗一样，最怕饭桶主导局势。而做为项目主负责人，你可以不是最懂项目的，但一定要清楚的是这个项目安排给谁去懂最合适，他有没有这个能力、是否能听从安排。一旦让他去懂，在项目推进过程中观察他是否能胜任、各子环节最懂的那个人又是谁。明白谁最懂、谁肯听众安排，则把相应部分的决定权交到相应最懂、最听从安排的人手中。而后除非出现特殊情况，自己不要去干涉、也不要让其他人去干涉。

D.主设计师

原则上讲，百张表内的数据库最好由一人来设计。一百张表，即使每张有一百个字段，总共也不过一万个，如果用工具生成，三五天内足够。当然数据库设计工作的难易并非是由表个数、字段个数决定的，而更多在于业务的复杂程度。表个数、字段个数在一定程度上可以反应出业务的复杂程度，但却非决定性因素。这里讲百张表内的设计工作量并不大，是说业务理清后将设计具体化成CDM、PDM的体力工作。要求由一人来设计是为了确保最终数据库的统一性、完整性、协调性，如果不能保证统一，最终项目的稳定性绝对得不到保障。不仅数据库的设计如此，架构、程序、前端、样式、脚本、UI都一样。在项目设计阶段，通常情况下，局部模块设计的优良并不会提升整个项目的质量，然而局部模块设计出现的问题最终却可能撼动整个项目的稳定。千里之堤毁于蚁穴，务必要确保设计工作的谨慎协调统一。再说大部分企业内部应用系统，项目规模有限，一个人主设计足够。

特殊情况下，如果项目大到一定程度，所有数据库设计工作交到一人手上着实过量，不得不安排多人参与其中，那主设计师也必须只有一位，且所有参与人员都要严格遵守相应的数据库设计规范。要利用PD的版本控制功能协调统一，最终由主设计师汇总校验所有人的设计，最终的数据库设计应该看起来像一个人的杰作，这也是程序、脚本、样式、UI设计开发所追求的目标。

主设计师汇总校验之后，还应再组织会议对设计成果校验，包括所有参与设计的人员、程序开发人员等，一起讨论。查找可能出现的不合理的地方，比如有部分可能和需求业务不合、可能会影响到开发实现等等。数据库的校验审核工作，参与人员尽可能多点，鼓励提问，有助于查缺补漏，发现问题。

E.关于DBA

有公司规定所有项目的数据库设计，DBA必须参与，但数据库设计工作重在对业务的把控、了解，其次才是对数据库本身的了解，这两项缺一不可，且前者更为重要。DBA的专业技能或许会更好一点，但是对业务的了解呢？还有公司的运维团队中，DBA会脱离项目实际，设置一些不必要的数据库规范，强加给开发团队。

在做项目时，一方面非常讨厌外部强加的规范，另一方面又不停的给自己、给团队设定规范。讨厌外部强加的规范是因为这些规范大都是些PMO指定的人凭空设想出来、脱离项目实际的，这些规范只会给项目带来更多的麻烦。而给自己和团队设定规范，则是为了约束设计和开发行为，确保项目最终实现的合理统一协调，这在后面的“命名规范”——“引言”中还有进一步的说明。

大的项目、大的团队可能会有多个专业DBA负责数据库维护工作，但就自己接触，即便公司、团队本身有DBA，绝大部分规模的项目中数据库维护工作还是多由程序开发人员兼顾。再者，真正优秀的DBA少之又少，有时不够专业的DBA过多介入反而会阻碍项目的正常开发。

DBA出现的时机应该是在开发人员无法解决数据库出现的问题时，比如当数据量大到一定程度，项目运行缓慢，仅凭程序优化已遇到瓶颈，这时可以向公司申请DBA介入，优化数据库设计、SQL语句等。再比如遇到数据备份问题、出现数据丢失问题等，也可申请DBA协助。如果项目规模大到一定程度，出现性能瓶颈问题是很正常的，这时专业DBA的作用才开始突显，且会在其中扮演一个非常重要的角色。

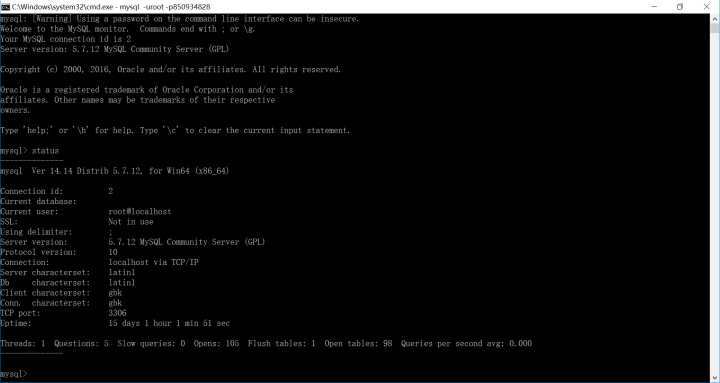
总之，DBA应该是在被需要时出现，而不应该被强制需要。

我的项目开发思路非常明确，关于团队，人越少越优秀越好，人员明确分工；关于开发模式，前端、后端、数据库明确分工。在项目规模、工作量允许的范围内人越少越好、涉及合作的部门越少越好，以便统一管理控制，节约沟通协调的成本。开发过程中的规范和约定采用大一统的方式，严格限制脱离中央管控的脚本或代码出现。

**3、基本原则**

A.数据编码

这里建议将MySQL数据库编码设置为utf8， SQLServer、Oralce可先采用默认设置，有需要则根据实际情况做相应变动。如果是MySQL在安装时就应注意，下图是当前电脑中数据库的编码设置，安装时没有改过，是默认的，所以显示的是latin1。



编码问题设置不妥当，数据库导入导出备份很容易出现乱码，尤其是旧版本的MySQL数据库。日常的数据库维护操作推荐使用其官方工具MySQL Workbench，Oracle的话推荐SQL Developer，也是官方工具。如果使用MySQL Workbench创建新的数据库，会要求选择Collation，默认即可。Collation的字面意思是字符序，用于指定数据集如何排序，以及字符串的比对规则。这地方的设置会影响到数据查询时大小写敏感的问题，这里简单一提，具体介绍的会在命名规范中给出。

如果是MySQL数据库，表类型统一使用InnoDB。InnoDB和MyISAM是在使用MySQL最常用的两个表类型，各有优缺点，视具体应用而定。其区别在于MyISAM类型不支持事务处理等高级处理，而InnoDB类型支持。MyISAM类型的表强调的是性能，其执行数度比InnoDB类型更快，但是不提供事务支持，而InnoDB提供事务支持及外部键等高级数据库功能。没有进行过额外设置，数据库默认安装的就是InnoDB类型。

B.高级功能

在本章节的引言中提到过，存储过程、触发器、函数、视图、事件等高级功能的设计规范都未给出详细说明，因为本规范中要求，在当前阶段的所有项目中，禁止使用存储过程、触发器、函数、视图、事件等高级功能，数据库中只有表，数据库只做数据存储、标明业务关系。而具体业务逻辑相关的处理均由程序实现，比如分页、日志记录等等。

某些情况下，一些业务逻辑放在数据库中处理比放在程序中处理要简单许多，但这会额外增加数据库的维护工作，从全局来看，也就是增加增个项目的维护工作。尤其是项目要求支持多数据库时，不同数据库存储过程、触发器、函数等的编写方式差异巨大，很难统一控制。业务逻辑下压到数据库的部分越多，这种维护就愈加困难，到一定程度必然要求有专业的DBA配合，额外增加了不必要的麻烦。视图的添加更是没有必要，连接查询可以实现的功能，用视图只会从全局上扰乱原有的数据库表关系，让程序多维护一套查询程序，徒增麻烦，而所得收益完全可由单纯的程序实现。

前面已提到所坚持的开发模式，给前端、程序、数据库明确的分工：前端只负责数展示、人机交互，由前端脚本处理展示交互相关的逻辑；后台程序负责桥接前端界面和数据库，处理核心业务逻辑；数据库部分只用来数据存储、标明业务关系。三部分各司其职，不要在数据库中直接处理业务逻辑，不要在后台程序中直接处理前端展示或人机交互的逻辑。

当项目庞大到一定程度、业务逻辑复杂到一定程度、数据量并发访问量增加到一定程度，单纯的从程序、前端中优化效果有限，需要多管齐下时，再来动数据库。即便到那时，最好也是在PD中统一设计这些功能，可以自由导入。否则，如果是多数据库支持的项目，每种数据库都得分别手动编写不同的代码，这样配置会很麻烦，也很难统一控制。而对于绝大部分中小型项目，这些麻烦本是没有必要出现的。

C.遵守范式

目前关系数据库有六种范式：

第一范式（1NF）属性不可分。是指在关系模型中，对域添加的一个规范要求，所有的域都应该是原子性的，即数据库表的每一列都是不可分割的原子数据项。

第二范式（2NF）在1NF的基础上，非码属性必须完全依赖于候选码（在1NF基础上消除非主属性对主码的部分函数依赖）

第三范式（3NF）在1NF基础上，任何非主属性不依赖于其它非主属性（在2NF基础上消除传递依赖）

巴斯-科德范式（BCNF）Boyce-Codd Normal Form（巴斯-科德范式）在1NF基础上，任何非主属性不能对主键子集依赖（在3NF基础上消除对主码子集的依赖）

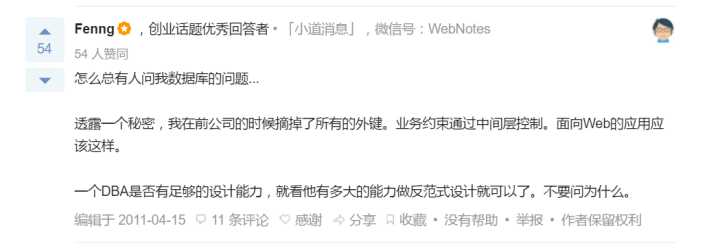
第四范式（4NF），要求把同一表内的多对多关系删除。

第五范式（5NF），又称完美范式，从最终结构重新建立原始结构。

这些范式仔细讲起来很耗文笔，关于范式的详细介绍可以参考文档末尾的参考文献。在设计数据库时，尽可能做到前四个即可。但范式并非硬性要求，而是设计时尽可能遵守的大原则。对于这些范式也没必要刻意去记，关系型数据库说到底只是二维的行列组合，参与的数据库设计多了，设计理念自然成型，会不自觉的去向这些范式中靠拢，设计过程中可以以此为依据校验自己的设计是否合理，及时修正不当设计。

特殊地方，为了操作方便，在不影响核心业务的情况下，允许小范围的不遵守范式。平时的多图片上传功能，可能只设计一个字段存储图片名称，这样字段值中就会包含多个图片的名称，里面用|或其它符号分隔，这样属性值就可分了，也就违反了第一范式——属性不可分；还有地区表中的，地区编码字段也隐含有等级信息；再就是菜单相关表，里面分等级互相引用等等；严格来讲这些都是不符合规范的。

在小范围内，不影响到核心业务流程，为了设计开发的方便，允许一些特殊的设计出现。规范是用来约束设计的，约束的目的是为了最终整体设计的合理性、为了最终项目的稳定性，如果这种全局的规范约束在局部影响到开发实现，可以适当调整，但前提是设计者要有足够的经验和能力驾驭住这种不当调整。



D.连接查询

因为不允许使用视图，所以在表设计时要尽可能的考虑到后面的连接查询，目前所接触项目的数据量都尚在可控范围之内，一般的业务系统，允许三张表以内的连接查询，五张表以内的要限制，超过五张的禁止。禁止使用子查询，子查询转换成关联查询。

如果要尽可能避免多表连接查询，那在设计数据库时，有关联关系的地方，一般从表中除了有引用主表主键的外键字段外，还要有一个或多个字段存放主表中的关键信息，比如病人表中有所属医院所属科室主键的外键字段，但还可能会有所属医院名称所属科室名称的字段。因为嫌数据冗余、维护不易，之前自己一般不设计除外键外存放主表信息的其它字段，但这样查询时又会多添好些麻烦。可如果加上这些字段，不单单数据冗余、维护麻烦，也不好保证数据的准确协调统一性。比如如果医院的名称被修改了，那按常理病人表中的医院名称也得做相应的修改才可以，这样，如果医院表被许多表引用，那就得对所有的表执行修改动作，很是麻烦，实现起来也不现实。如果都加上引用约束，依赖数据库自己的关联自动更新，觉得也不是很好，影响程序执行速度且不易维护。可如果是在会诊单表（类似于订单表的功能）中出现类似的情况，就无须有这种顾虑，如果这个会诊单在产生时医院是这个名称，后面名称有了更改，那会诊单中的医院名称还是显示早前的即可，无须做相应更改，这也是符合逻辑的。

类似情况经常出现在字典相关信息的存取中，平时只在表中存字典编码，但查询时又往往要求同时提供字典文本，核心业务表中的字典字段往往比较多，暂时没有好的方法一次高效提取完整信息。后面在做类似设计时，核心业务表，字典字段比较多的，根据实际情况，考虑同时存入字典编码和字典文本，这样可避免部分连接查询。但同时，还是会出现上面提到的问题，一方面是数据冗余，一方面业务表中的字典文本有可能会和字典表中的文本不一致——如果字典信息有更改的话。

这里很难找到一种两全其美的办法，既能避免数据冗余，又能让程序在执行读和写动作时都方便高效，如何在各方面之间拿捏均衡是个人经验问题。通常情况下，是在数据的协调性、准确性允许，跨表查询又不容易（从表外键较多）的业务模块，采用在从表中附加额外字段的处理方式；在对数据显示的同步性、准确性有严格要求，跨表查询也相对容易（从表外键较少）的业务模块，采用从表中不设外键以外的附加字段、而使用关联查询的方式获取完整数据信息。

在普通的业务系统之外，还有一种情况，可能不得不大量使用函数、子查询、嵌套查询。和可视化部门合作过的一些项目，系统前端引入ECharts，大量的环形图、柱状图、折线图等用于展现数据统计分析结果。有些图形需要的数据很难用简单的SQL一次提取出来，可如果多次提取后再由程序组装处理，又太过麻烦，最后还得考验SQL，此种情况是允许子查询、嵌套查询、多表连接等复杂SQL出现的。



过去曾借助外部框架设计出一套异常灵活的架构，封装了查询对象，消除掉了SQL语句，应付通用的业务系统足够，但在做类似这种统计分析功能时却变得各种不自在，到后来还是觉得直接写SQL更方便。此处只能在设计的架构上放开一个口，让开发人员可以自由编写SQL语句提取数据，最终的查询结果统一封装成一个List<Map<String, Object>>，然后交由程序自动序列化成JSON格式（包含多个对象的数组）返回给前端。

E.主键外键

每张表必须有唯一主键，此主键只用来标识记录唯一，没有其它任何功能，业务表中其它任何的惟一字段均不能做主键，比如用户表中的身份证号、地区表中的编码，虽然也是唯一，但均不可做主键。主键为UUID，统一命名为id，统一为char(32)类型。相较于自增主键类型，使用UUID会导致查询速度稍慢，可用自增主键的话，如果同时支持多数据库就比较麻烦了，比如Oracle实现自增功能必须用序列辅助处理，还有，备份合并不同数据库中相同表数据也会出现冲突，设计分布式架构、横向拆分表也会出现类似问题等等。关于主键，还有一点要强调，原则上讲，多对多关系中间表外的任何其它表不允许出现复合主键。

外键，多和主键对应，多是引用另外一个表的主键，那么这个外键及所有表的主键的类型也应该是一样的，都规定成char(32)类型。再就是外键的命名也要统一，这个也可以通过PD由CDM转换成PDM时统一处理，使用“主表名\_主键名”的命名方式，比如user\_id，如果一个从表中有多个字段同时引用主表的同一个字段，那再用其它标识，比如在会诊单申请表中会诊发起医院（sender\_hopital\_id）和会诊接收医院（receiver\_hospital\_id）。

主键和外键是数据库设计中非常关键的两个地方，在后面的约束控制和命名规范中还有更详细的介绍。

三种数据库关系中，多对多关系是最麻烦的，中间表往往使用复合主键，在程序架构上对这部分业务的处理不太容易，设计程序架构时要特别注意。

F.逻辑删除

关于针对数据库的删除动作，仔细考虑后还是决定，现阶段暂时仍旧使用物理删除。严格意义上来讲，所有的软件项目都不应该有物理删除动作，目的很明确，就是为了数据的安全性完整性。如果使用逻辑删除，给每张数据库表加上一个is\_active字段，数据的安全性完整性虽然可以得到保障，但却跟程序编写带来许多麻烦。比如所有的查询方法都需要额外加上is\_active=’0’的判断条件，在涉及多表连接查询时就很不方便。再有，执行假删除动作时的级联处理也会相对麻烦。还有，如果数据表的某个字段要求唯一，并强制约束，比如用户表中的登录用户名字段，设计为逻辑删除的话，一旦有新的同用户名记录就无法插入。但如果不将该字段设置为唯一性约束的，那么在每次插入数据的时候，都需先进行一次查询，看看有无未（逻辑）删除的同名记录存在，低效率是一回事，而且在高并发的系统中，很难保证其正确性。

此外，使用逻辑删除，随着项目应用时间的增加，将会有大量无效数据产生，如何处理？如果数据量大到影响查询速度，程序如何做优化？像日志表，很容易就到上百万。如果定期将逻辑删除的数据转移到另外的表中，觉得处理起来也不简单，还可能会有附加问题出现。如果使用脚本定期彻底删除逻辑删除的数据，如何叠加备份这些删除的数据？

综合考虑，对于现阶段的项目，逻辑删除所带来的好处有限，但带来的问题却很多。如果平时做好数据备份工作，还是可以预防物理删除隐患的。但心里应该清除，当项目大到一定程度，对数据安全性的要求高到一定程度，使用逻辑删除代替物理删除是必然的，在后面的数据库设计中，可以先小范围的尝试使用逻辑删除，一旦开发模式成熟，就全面使用逻辑删除代替物理删除。

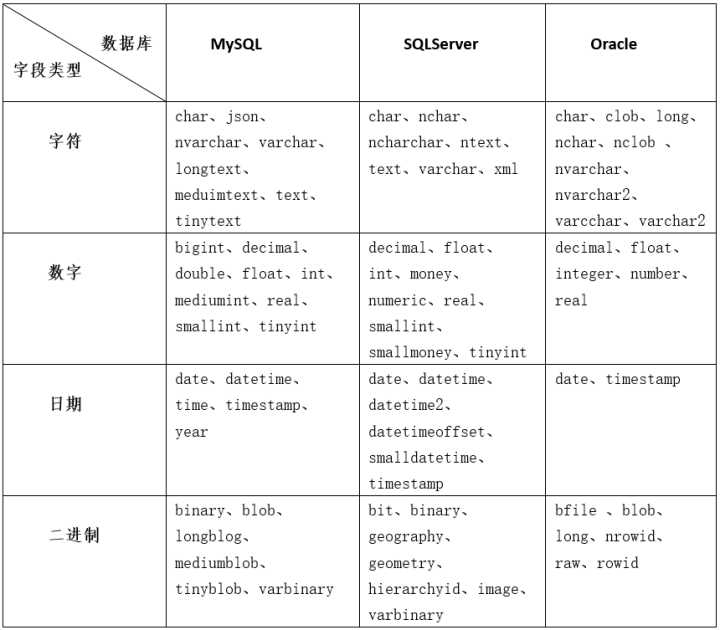
G.性能安全

对于中小型项目，数据库设计过程中过多的考虑性能问题往往弊大于利，程序部分的设计开发也是如此。不是不考虑，而是不要过多考虑。性能及安全问题太大，这些问题的重要性都是随着项目规模增长的，在中小型项目中过分的考虑只会给开发带来不必要的麻烦。中小型团队，时间人力成本有限，要用在关键环节上，用最短的时间做出成型的项目是最重要的。至于设计开发，尽可能保证好其协调统一灵活稳定性，一旦出现问题或需求调整，可以随时灵活的更改就好。

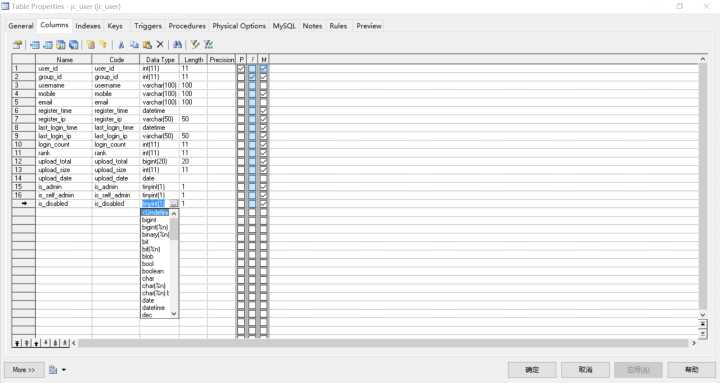
**4、字段设置**

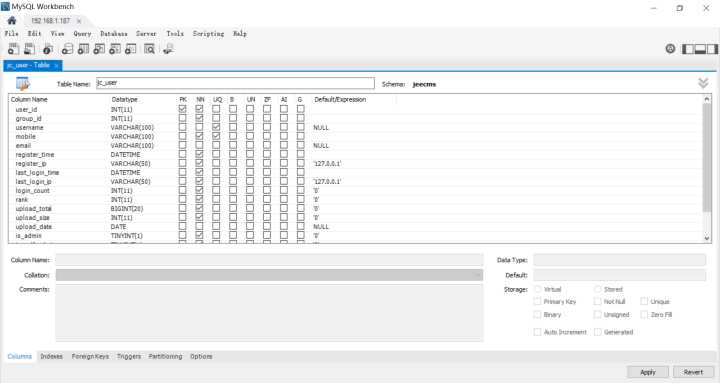
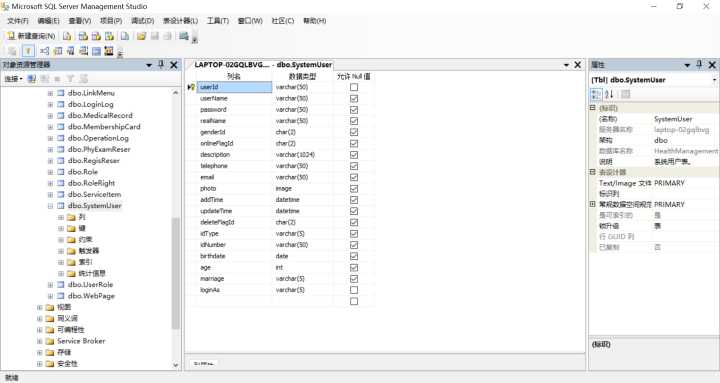
A.字段类型介绍

不同的关系型数据库在字段类型的具体化上差异较多，这里无法一一详述，但具体化的字段类型再多，无外乎几种：字符、数字、日期、二进制。



下面有三张截图，前两张是一张典型的用户表结构，上面的是PD中的PDM，下面的是导入到MySQL的具体表结构，第三张截图是SQLServer数据库中的一张用户表的结构图：



各个数据库的某个具体类型，这里无法详述其适用情况，本模块只制定通用的字段设置规范。

B.不为空默认值

表中应该尽可能避免可为NULL的列，且尽可能显示设置默认值，尤其是被索引的列。

在MySQL数据库中，空值是不占用空间的，而NULL其实是占用空间的。再者，MySQL表的列中包含NULL的话，该列就不会包含在索引中，也就是说使用索引是无效的，现在不确定其它数据库是否也是如此。 所以考虑今后可能会使用索引的字段，就要设置字段属性是NOT NULL。比如，如果某个字段后面可能会作为查询关键字使用LIKE的形式进行搜索，就要将该字段定义成索引以提高查询速度，那这个字段属性就是NOT NULL的。

除以下数据类型的字段外：timestamp、image、datetime、smalldatetime、uniqueidentifier、 binary、sql\_variant、binary 、varbinary，表字段应尽可能显示设置默认值。建议数值型的默认值为数值0，布尔型的默认值为数值1（通常情况下，系统中所有逻辑型中数值0表示为“真”、“正常的”；数值1表示为“假”、“异常的”，这种编码后面还会有介绍），datetime、smalldatetime类型的字段没有默认值，必须为NULL。

如果数据库中某个字段有默认值，那么觉得在程序开发过程中，对应实体类的属性应该设置同样的初始化值才合理，记得动软代码生成工具中的框架就是这样设置。之前自己的程序设计中没有注意到这点，自动生成的所有实体类的属性都没有默认值。

注意区分NULL和空字符串是不同的，数值型字段中NULL和0更是两码事。如果在数据库设计过程中不允许出现NULL字段还好，但如果有允许NULL而没有设置默认值的字符型字段，程序对记录执行了写空字符串动作和压根未执行写动作是两码事；如果有允许NULL且设置默认值为空字符串的字段，则无法做这种区分。当然，通常情况下，我们认为文本框中空字符串的提交动作等同于未执行写入。有些类似的情况是，在程序开发中，一个空的List对象，或者一个new出来的空对象，和NULL也是不同的，要注意。

像订单（会诊单）这种表，取消、退回、安排这些字段的信息都不是必须有的，可以分流到子表中存储，放在一张表中会导致出现很多可为NULL或空值的列。之前并不赞成这种过分分流的方法，因为这会另信息的维护变得麻烦，如再有类似情况，应该根据实际综合判断取舍。也要在设计时尽可能遵守第二、三范式，非主属性完全依赖于码（主键）、消除传递依赖，不要让某张表过分臃肿。

C.类型长度设置

当字段定义为字符串类型时建议使用varchar而不用nvarchar以节省空间，通常情况下，都要用尽量少的存储空间来存储一个字段的数据，能用int类型的就不用char类型，能用char类型就不用varchar类型，能用varchar(20)的就不用varchar(25) 。char和varchar长度设计需要根据业务实际需要进行长度控制，禁止预留过长空间。比如主键要求用UUID，那就统一为char(32)，可以固定的部分就都固定下来。varchar类型虽然根据实际长度进行存储，但内存分配则是根据指定长度，不合理的长度设计会导致内存的不合理占用。

varchar是变长存储，字段长度是数据库一种约束，定义合理的长度也可以让人容易理解字段的用途。MySQL中定义的长度如果小于255，字段长度用1个字节表示，如果超过255，字段的长度将固定用2个字节表示。Oracle没有这样的问题。字段定义的长度对索引也有较大影响，MySQL数据库索引存储的长度都是定义的长度，不是实际字符的长度，这是一个非常大的问题，估计主要原因是为了实现简单，所以MySQL在索引上会浪费大量的空间保存字符串。

前台、程序以及数据库各部分之间对字段大小的限制务必处理恰当，为了节省存储空间，选取的数据库字段容量在一定范围内应该尽可能小，而为了对程序提供更好的扩展支持，又应该尽可能的设置大些，具体字段类型、字段长度如何设置，根据实际情况取得均衡。而后台程序部分，对数值大小长度应该做好校验处理，确保插入数据库的值大小长度不要超过限制。同时前端也应该给出明确的校验提醒，让用户按提示输入，决不允许不提醒用户而擅自把数据处理后插入数据库中（这种错误真有人犯过）。这样，前端、程序、数据库全方位校验处理，自然可以保证数值的准备性、存取的合理性。

除非要保存文章内容， text字段尽量少用，如果要用能拆到冗余表中最好。禁止使用blob类型保存大文本、附件、图片等，对于图片、文档等附件数据库中只保留原始文件名和存储路径。网上也有建议使用其他存储方式的，比如TFS、SFS等，可以参考。

禁止使用float、double类型，建议使用decimal替代。decimal(a,b) ，a指定指定小数点左边和右边可以存储的十进制数字的最大个数，最大精度38。 b指定小数点右边可以存储的十进制数字的最大个数。小数位数必须是从 0 到 a之间的值。默认小数位数是 0。比如decimal(5,2)规定了存储的值将不会超过5位数字，并且小数点后面有2位数字。

D.Oracle的CLOB

在Oracle中，CHAR 为定长字符串，最长2000字节 。VARCHAR2 为变长字符串，最长4000字节。NCHAR和NVARCHAR2分别与CHAR和VARCHAR2相对应，但存储的数据为 NLS字符 。

目前VARCHAR是VARCHAR2的同义词。工业标准的VARCHAR类型可以存储空字符串，但是Oracle不这样做，尽管它保留以后这样做的权利。Oracle自己开发了一个数据类型VARCHAR2，这个类型不是一个标准的VARCHAR，它将数据库中VARCHAR类型的列可以存储空字符串的特性改为存储NULL值。如果你想有向后兼容的能力，Oracle建议使用VARCHAR2而不是VARCHAR。

在Oracle中没有TEXT类型，但有用于大文本存储的CLOB类型。Clob是指大字符对象，也就是英文Character Large Object的缩写；Blob是指二进制大对象，也就是英文Binary Large Object的缩写；由此可见这两个类型都是用来存储大量数据而设计的。

LONG 最大存储2G字符数据，但现在已不推荐使用（改用CLOB）；CLOB 在Oracle 9i 及以前，最大可存储4G字符数据 ，在Oracle10g及以后，最大可存储4G\*数据库块大小的字符数据；NCLOB 基本同CLOB，就是存储的数据为NLS。

在Oracle数据库表中使用CLOB类型字段，最大的问题是备份数据时不好处理。在有些情况下，给政府、企业做项目，只给你Oracle的访问权限，而不给你Oracle所在服务器的操作权限，也就是说自己无法操作Oracle服务端工具。但Oracle的客户端中又没有exp、expdp命令，这样备份导出数据库就不好弄了（此处不提沟通协调甲方处理）。SQL Developer是Oracle的官方工具，用其导出数据库，如果导出的是SQL格式，那CLOB类型字段的数据将直接被忽略——这绝对是无法接受的。官网上有文章说可以将数据库导出为loader或pdf格式，自己尝试导出这两种格式，发现不能导出成单个文件，会导出很多的文件。而且导入时也需要用到额外的工具——Oracle服务端的sqlldr.exe，这样只借助客户端也是不行的。

在SQL Developer“工具”菜单下，还有两个选项：“数据库Diff”及“数据库复制”，如果所处网络可同时访问源数据库和目标数据库，可用这种方法互相拷贝数据，但是同样的问题，这种数据库复制方法，仍然是不能处理COLB、BLOB的字段。而且我发现，凡是带有这两种字段的表，在复制时都没有数据，不是相应字段没有数据、是整个表的数据都没有复制，其它没有BLOB、CLOB字段的表，数据拷贝都正常。

也曾想使用的PL/SQL Developer工具进行备份，导出了PL/SQL Developer自己的格式（pde）。可是却提示stream read error，到网上一查，原来PL/SQL Developer自己的格式也是不支持COLB 、BLOB类型字段的导出的。

之前同事介绍过Navicat for Oralce工具，但其在导出CLOB、BLOB类型的字段时，如果字段中的数据过长，也是无法再正常导入的。这个小工具看似简单轻巧，在执行一些操作时问题却很多，不宜作为一款常用的Oracle管理工具。

这样看来，只有expdp命令才能有效导出clob、blob格式的字段了。

cd D:\app\bsc\product\12.1.0\dbhome\_1\BIN\

exp username/password@tnsname file=C:\Users\Administrator\Desktop\20170323.dmp owner=username

imp username/password @tnsname file=C:\Users\Administrator\Desktop\tnsname.dmp ignore=y full=y statistics=none

不过，如果你虽然没有源数据库服务端的访问权限，却有目标数据库服务端的访问权限，且两个库可在一个网络中访问，也是有办法用EXP命令备份源数据库的。就是让目标数据库服务端的TNS监听源数据库的实例，再利用目标数据库Oracle服务端的exp.exe工具远程导出源数据库，导出导入命令和上面类似：

cd D:\app\bsc\product\12.1.0\dbhome\_1\BIN\

exp origin\_username/origin\_password@origin\_tnsname file=C:\Users\Administrator\Desktop\20150508.dmp owner=origin\_username

imp target\_username/target\_password@target\_tnsname file=C:\Users\Administrator\Desktop\20150508.dmp ignore=y full=y statistics=none

此外，用PL/SQL Developer工具备份数据时，Export User Objects菜单命令导出的是SQL文件，在这里你可以将建表、序列、触发器、存储过程等的SQL语句全部导出成一个文件，但是这里面并不包括数据。要想导出数据，必须用Export Tables……菜单命令，导出DMP文件。当然也可以导出其它格式的文件（SQL、PDE），但建议用DMP格式，因为前面已经说过，如果表中有CLOB类型字段的话，用其它格式的导出方式恐怕有问题。

如果一个表不存在，而这个表中没有CLOB、BLOB这种特殊数据类型的字段，用DMP导入数据时PL/SQL会自动建立这个表。但如果一个表不存在，而这个表中又有CLOB、BLOB这种特殊字段，直接导入DMP格式的文件会报错 IMP-00003: 遇到 ORACLE 错误 959。所以在Oracle中导入数据库时应该先执行用Export User Objects导出的SQL文件，这样相关的序列、触发器、表结构都已经建好了，再导入用Export Tables…… 导出的DMP文件，也就是导入其中的数据，就万全了。

E.数值类型选择

float：浮点型，含字节数为4，32bit，数值范围为-3.4E38~3.4E38（7个有效位）

double：双精度实型，含字节数为8，64bit数值范围-1.7E308~1.7E308（15个有效位）

decimal：数字型，128bit，不存在精度损失，常用于银行帐目计算。（28个有效位）

//结果显示为345.9876，只显示7个有效位，对最后一位数四舍五入。

float f = 345.98756f;

//结果显示为345.975423578631，只显示15个有效位，对最后一位四舍五入。

double d=345.975423578631442d;

//可以支持28位，对最后一位四舍五入。

decimal dd=345.545454879……

float和double的相乘操作，数字溢出不会报错，会有精度的损失。当对decimal类型进行操作时，数值会因溢出而且报错。

Oracle中的数值类型，Oracle只是在语法上支持decimal类型，但是在底层实际上它就是NUMBER类型，支持decimal类型是为了能把数据从Oracle数据库移到其他数据库中(如MySQL、DB2等)。Oracle的NUMBER数据类型的精度：NUMBER(P,S) ，P： 1---38， S：-84---127。S代表的是小数位数，P代表的是总位数（整数位数和小数位数）。所以，平时如果在Oracle中用自增主键，长度设为NUMBER(10)的话，相当于NUMBER(10,0)，表示最高可记录到十亿级的数据量。

下图是MySQL中的整数型数值类型详述：



F.通用字段处理

日期时间类型字段，网上有建议，采用int来记录unix\_timestamp，自己还是习惯用datetime。不过设计原则是粒度越小越好，所以这里要求日期时间类型的字段，尽可能精确到时分秒，用datetime类型。即便是像生日（birth\_date）这种字段，一般只存储到年月日，但在选择字段类型时建议还是用datetime而非date，以防万一。如有部分时间字段着实无须记录到时分秒，则用date类型。严禁使用varchar等字符串类型记录日期时间，更不要把时间猜分，年在单独的字段、月在单独的字段、日又是单独字段，老实讲TM想不明白这种人的设计思路是什么样的。

网络IP字段，网上有建议，除特殊情况一律用bigint来记录inet\_aton值，但这种存储方式貌似只在MySQL中适用，这里要求还是用varchar存储。关于inet\_aton想了解的话可以看下参考文献中的“MySQL的IP处理函数inet\_aton()和inet\_ntoa() ”。

字典编码字段，之前在SQLServer中设计数据库时统一使用char(2)类型，Oracle数据库中统一使用number(2)，在MySQL中统一使用tinyint(2)。现在想来最合理的还是设置为tinyint(2)，以后数据库字典编码字段统一按此设置。就是Oracle中没有tinyint类型，不知道如果在PD中设置此种类型，导入到Oracle时会自动转换处理还是直接报错。

备注字段，尽可能在所有表中都保留这个字段，也是给前端信息录入预留一个可扩展部分。统一命名为remark，字段类型为varchar(200)，最多100个中文字符。再多的话说明有额外信息，就不适合放在备注字段中了，要再加新字段存储。

排序字段，不是每个表中都需要额外的排序字段，但有些表这必须有，比如记录菜单信息的表、门户网站中存放文章内容的表等。这里推荐统一使用int(10)做为所有表中的排序字段类型。

字段设置部分撰述内容较多，相对详细，这是比较重要的一部分。以后的数据库设计，字段类型选择、字段长度设置部分都要以此为依据。

**5、相关注释**

数据库是整个项目的地基，后面程序架构设计及具体开发工作都要在此基础上进行，如果没有注释或者注释混乱，会给开发者带来很多困扰，无形中影响项目进度。

在相对成熟的技术团队中，所用开发模式、架构、框架也都已经成型，如果接手的项目类型不超出平时常规的范畴，以往的开发模式、架构及所用框架等都是无须做大的变动即可拿来复用的。这样，在数据库设计过程中或完成之后，程序的具体设计开始，再到进入正式开发过程中，需要频繁组织开发人员参与的会议。会议讨论的都是具体的开发工作，数据库文档是参考依据、必须的会议资料。

数据库文档一般根据PD中的数据库文档模板生成，PD自带的数据库文档模板太杂乱，所以最好单独设计一套适合自己团队使用的模板。比如平时在数据库设计过程中不用存触过程、触发器、序列、函数等高级功能，那我设计的模板自然不包括展示这些信息，只显示表结构、约束、表注释、字段注释等信息，生成的文档也会简单明了。如果不想单独设计，网上也有一些他人整理的模板，或可根据实际情况拿来使用。

团队内部使用的数据库文档一般是HTML格式，查看比较方便，而对外提供的，向PMO或客户提交留存的，一般是RTF或DOC格式的。

用PD完成设计之后，将PDM导入到数据库，同时生成数据库文档，后期所有针对数据库的讨论都会以这份数据库文档和PDM图为依据。后面数据库表结构如有变化，文档会同步更新。而这份文档中的文字说明内容，就来自于数据库设计过程中的相关注释。再有，现在的程序设计好之后，具体代码大都是由工具生成，程序中的实体类、方法说明等等最终也都是由数据库中的相关注释演化而成。所以注释说明的语气，甚至标点符号都要注意，只有数据库注释规范了，数据库文档、程序注释、程序文档、对外接口文档才能规范，因为这些的起始点都是在数据库中的注释上面。

数据库中的关键注释其实就两部分，一部分是表注释，一部分是表字段注释。表注释对应程序中实体类的注释，字段对应程序中实体类属性的注释。

A.表注释

对于表的注释，要求简单明了，先说明表中文名称，句号分隔，然后跟功能说明，无需特别说明的，则只说明中文名称即可。同时，为了最终生成代码注释的可读性，不允许在说明中出现“表”字眼。比如订单表，直接注释“订单”即可，如核心业务表需要其它特殊详细的功能说明，要考虑到最终生成程序实体类注释的可读性。再有，如果数据库由多人协作设计，那表注释中必须标明创建者和创建时间信息，示例如下：

创建者：孟宪志。创建时间：2016-09-10 13:30。说明：系统日志。用于记录用户对系统的所有操作。

数据库设计完成进入开发阶段，后续扩展新增的表也都要按如上格式注释，同时说明是后续新增表，及新增原因等。

B.主键字段

主键字段，无须特别注释，注释就两个字：主键。

C.外键字段

外键字段，注释先说明字段中文名称，句号分隔，后面要特别说明是外键，然后说明引用哪个表的哪个字段，再句号分隔，如还有其它说明，则继续跟后面。比如：

` applicant\_doctor\_id` char(32) DEFAULT NULL COMMENT '申请医生。外键，引用医生表（doctor）的主键（id）。理论上可填多个，以“|”分隔'。

前面说过为了避免多表连接查询，有关联关系的地方，一般从表中除了有引用主表主键的外键字段外，还要有一个或多个扩展字段存放主表中的关键信息，这个扩展字段通常都是和对应外键字段并列存放，注释时先说明字段中文名称，句号分隔，后面要特别说明是名称（或其它），然后说明和哪个表的哪个字段对应，再句号分隔，如还有其它说明，则继续跟后面。格式如下：

` applicant\_doctor\_name` char(32) DEFAULT NULL COMMENT '申请医生。名称，和医生表（doctor）的医生名称字段（name）对应。理论上可填多个，以“|”分隔'。

D.字典字段

字典字段在数据库表中出现非常频繁，要求所有表中的字典字段注释也要统一格式。注释先说明字段中文名称，句号分隔，后面要特别说明是编码，对应哪个字典表的哪个字段，再句号分隔，后面跟详细说明，如果字典项不多的话，要求先枚举说明每种字典编码代表的不同意义，句号分隔，如还有其它说明，则继续跟后面。比如：

`status\_code` varchar(5) DEFAULT NULL COMMENT '转诊单状态。编码，对应数据字典表（dictionary）中的编码字段（code）。目前先定义6个状态：01 已提交，02 已取消，03 已审请，04已安排，05 已拒绝，06 已结束。03是向HIS系统执行审请，04是HIS系统已安排入院日期，下转诊时这两个字段没有区别。'

如果有字典文本扩展字段（类似于外键的扩展信息字段），格式与上类似：

`status\_text` varchar(50) DEFAULT NULL COMMENT '转诊单状态。文本，对应数据字典表（dictionary）中的字典项名称字段（name）。目前先定义6个状态：01 已提交，02 已取消，03 已审请，04已安排，05 已拒绝，06 已结束。03是向HIS系统执行审请，04是HIS系统已安排入院日期，下转诊时这两个字段没有区别。'

字典表细分的话也有好多种类型，后面特别说明中还会细讲。除了数据库中的通用字典表，还有一些常见表，比如地区表、ICD字典等，也是一种字典表，其共性是：表结构设计完成后、程序开发前即完成数据填充，且后期很少改动；主键外至少要有编码和文本两个非空字段，如果字典表中有细分子项的话，还会有字典项所属类型字段。

E.普通字段

除以上三种常见类型的字段外，其它字段，都可用统一的方式注释。先说明字段中文名称，句号分隔，后面跟详细说明。比如：

`bed\_id` varchar(50) DEFAULT NULL COMMENT '入院床位号。HIS系统安排并反馈的入院床位号，如果是下转诊，此部分由医生手动填写'。

还要特别说明的是，如果项目进入开发阶段后需要对现有表新增某些字段，或对现有表的某些字段进行修改，和新增数据库表一样，也必须特别说明。

创建者：孟宪志。创建时间：2016-09-10 15:38 。说明：会员职务。交易平台新增字段，个人用户此字段为空。

相关注释部分，先说明了数据库中注释对项目开发的重要性，然后分五个子模块：表注释、主键字段、外键字段、字典字段、普通字段，详细介绍。如今后所有的数据库设计都能严格按此规范要求、按给出的格式添加注释，则相关数据库文档、程序文档、接口文档都可随之走向正规。

**6、约束控制**

建立约束的目的是为了防止数据库中出现不符合语义规定、不符合业务逻辑的数据，为了确保数据的完整性、合法性。如果数据具有完整性，则表示数据有效——正确并且准确、数据库的关系结构是完整的。完整性约束可以加强数据库的关系结构，这些规则使数据在各个表之间保持一致。

总的来说有五种约束类型：唯一性和主键约束、外键约束、检查约束、空值约束、默认值约束，对应五大关键词，UNIQUE和Primary Key, Foreign Key, CHECK, NOT NULL, DEFAULT。这五种约束类型，除检查约束外，其余的在前面的章节中都有过介绍，比如主键、外键字段类型的选择，字段长度的设计，注释的规范等等，在后面的命名规范中也还会提到。

检查约束（CHECK），用于检查列的类型和范围，语法：CONSTRAINT [constraint\_name] CHECK (condition); 比如：check(Age >2)。过去没有在数据库中添加检查约束的习惯，在本规范中，也暂不做此要求，这会增加数据库设计过程中顾虑点，还是交由程序和前端来控制这种细节上的数据合法性。

本规范中要求，除检查约束外，其余四种都要在数据库中加强约束，同时，程序也会根据数据库中约束的设置做相应处理。

主键不用说，一般情况下都会加约束的。要注意的是唯一字段，比如用户表中的电话、邮箱、登录账号等等明显唯一的信息，都要加强约束，还有多对多关系中间表里的复合主键，也要加强约束。再就是，前面也提到过，本规范中所有主键字段统一使用UUID，char(32)类型，具体命名方式后面也会讲。

外键部分，类型自然要和主键一样统一为char(32)，除了外键字段，为了保存主表其它信息的扩展字段类型也要和主表中的相应字段类型统一。过去参与的项目中有的在数据库中加了外键的强约束，有的则没有，有的是在用PD设计时加上了主从引用关系，但在导入到库中时去掉了外键的强约束，而只保留表结构。

表面看，数据库中加上外键、非空、唯一等强约束会给程序编写带来麻烦，实则不然。约束是为了保证数据的合理性，如果数据库设计的约束本身没问题，那程序编写中因约束而照成的不便就多是程序本身的不合理造成的。万千世界，所有约束规范都是为了确保被约束规范的对象将所参与的事情做的更好，这种最终的好不是只针对某个对象的，而是针对所有参与其中的。假如某种规范不是这样，那它本身就是不合理的；假如它是这样的，而被约束规范的对象觉得在这种约束下做事情不便，那就是他自己做事方法的问题了，要由他自己根据规范修正。

所以强约束带来的不便只幻象，这种不便不但不是坏事，还会倒逼程序开发更加趋于合理，修正开发中的错误。而不加强约束，程序可以肆无忌惮的对数据库进行任何操作时，如果架构设计师及开发人员的经验、技能较强还好，否则的话，程序很可能会因有意无意读写错误信息，造成针对数据库的无效或错误操作，破坏掉数据的完整性、合理性。

有些情况下，加不加强约束设计者是决定不了的，比如公司有通用的权限管理系统对所有子项目统一管理，那在设计子项目的数据库时就不能添加和用户相关的外键约束了。还有其它类似的情况：两个项目之间有逻辑交差，但数据库相互隔离，只能通过互相调用接口处理业务。这种情况下，如果想保证好数据的完整性准确性，只能由程序在业务逻辑层控制，必须设计开发足够精密才能做好。

空值约束和默认值约束不再多说，在前面字段设置中的NOT NULL子模块讲的很详细了，只要记得设置字段为NOT NULL、并给字段设置默认值可以提高查询速度、节省表空间就可以了。

**7、索引添加**

索引是一个单独的、物理的数据库结构，它是某个表中一列或若干列值的集合和相应的指向表中物理标识这些值的数据页的逻辑指针清单。数据库中索引的概念与书索引的概念非常类似，不同之处在于数据库索引用来在表中查找特定的行。

在各种类型的数据库中，索引又可细分为好多类型，比如之前系统整理过Oracle中的索引分类：非唯一索引（最常用）、唯一索引、位图索引、局部有前缀分区索引、局部无前缀分区索引、全局有前缀分区索引、 散列分区索引、基于函数的索引。Oracle存储索引的数据结构是B\*树，位图索引也是如此，只不过是叶子节点不同B\*数索引。

并未做过严格对比，不清楚添加索引后对查询速度的提升到底有多少，对插入删除修改的速度影响又有多少。就过去的经验，对于Oracle来讲，百万级数据量一般不会出现明显性能问题的，无需做特别的优化；MySQL的话，超过五十万条数据就有点卡了，超过一百万就得做优化了。对于诸多中小型的企业内部应用系统，如果数据量未达到这个级别而出现查询性能问题的，多是因为程序，开发模式、架构、框架、编码细节或SQL语句中肯定有特别不合理的地方。所以对于当前阶段的项目，在出现查询性能问题之前，不建议在数据库设计过程中手动添加索引，而且大多数数据库是会自动给主键和有唯一约束的字段添加索引的。

当数据量达到一定程度，仅通过程序优化对性能提升有限时，可以考虑同时对数据库进行调校，这时会优先考虑建立索引。

在前面的字段设置中的NOT NULL子模块提到过NULL值对索引的影响，下面再列出一些索引使用注意事项，大都是从网络上整理出、自己认为比较合理的，如果要建立索引，尽可能遵循这些规范（第一条没大看明白）：

a． 逻辑主键使用唯一的成组索引，对系统键（作为存储过程）采用唯一的非成组索引，对任何外键列采用非成组索引。考虑数据库的空间有多大，表如何进行访问，还有这些访问是否主要用作读写。

b． 大多数数据库都索引自动创建的主键字段,但是可别忘了索引外键，它们也是经常使用的键，比如运行查询显示主表和所有关联表的某条记录就用得上。在经常进行连接，但是没有指定为外键的列上也要建立索引，而不经常连接的字段则由优化器自动生成索引。

c． 在频繁进行排序或分组（即进行 GROUP BY 或 ORDER BY 操作）的列上建立索引。

d． 不要索引blob/text 等字段，不要索引大型字段（有很多字符），这样作会让索引占用太多的存储空间。

e． 不要索引常用的小型表，不要为小型数据表设置任何键，假如它们经常有插入和删除操作就更别这样做了，因为对这些插入和删除操作的索引维护可能比扫描表空间消耗更多的时间。

f． 在条件表达式中经常用到的不同值较多的列上建立检索，在不同值少的列上不要建立索引。比如在雇员表的“性别”列上只有“男”与“女”两个不同值，因此就无必要建立索引，因为建立索引不但不会提高查询效率，反而会严重降低更新速度。

g． 如果待排序的列有多个，可以在这些列上建立复合索引（Compound Index）。

h． 索引不是越多越好，添加或删除表记录时，必须花费额外的时间来更新该行的索引。读的方便，写的就会有麻烦，反之，亦然。

**8、特别说明**

A.数据字典表

通用字典表的表结构如下：

/\*==============================================================\*/

/\* Table: Dictionary \*/

/\*==============================================================\*/

CREATE TABLE [dbo].[Dictionary](

[dictId] [varchar](50) NOT NULL,

[dictTypeId] [varchar](50) NOT NULL,

[code] [varchar](50) NULL,

[name] [varchar](200) NULL,

[remark] [varchar](1024) NULL

)

/\*==============================================================\*/

/\* Table: DictionaryType \*/

/\*==============================================================\*/

CREATE TABLE [dbo].[DictionaryType](

[dictTypeId] [varchar](50) NOT NULL,

[name] [varchar](50) NULL,

[remark] [varchar](1024) NULL

)

过去接手项目的字典表大都是这样设计，在找到更优设计之前，字典部分延续使用此结构。不建议在通用字典表中分等级，如果有要分级的字典数据，建议额外建表处理，比如地区字典，或者是左侧菜单（严格来讲，左侧菜单已经算不上字典）。

网上有建议将字典项放到XML文件中维护，如果是某些特定的字典项，比如刚提到的菜单项，数据量较少还可以，超过一定量就不现实了，对于大多数业务系统，通用字典表是必须有的。

自己对于字典数据的定义是，可枚举，系统运行前即被录入、运行后除非有特别权限出现特别情况不可随意更改添加删除。除了通用字典表外，全国省市县地区信息表、医疗系统中存放ICD数据的表等，原则上来讲也是一种字典表。

关于布尔型字段，比如男女、是否、对错等，过去的设计中没有使用数据库特定布尔类型定义的习惯，都是放在字典表中统一定义，后面的设计中也要求如此。对于编码，习惯用0和1，编码规则通常是：0是多数的、常见的、正确的、主要的、正常的，1是少数的、少见的、不正确的、次要的、不正常的。比如是否中0代表是、1代表否，对错中0代表对、1代表错，男女中0代表男、1代表女，建议布尔型字段的字典编码延续此编码方式。

虽然在业务表中的相关字典字段存编码已经成为自己的一种习惯，但和同事协作时发现还有人直接存文本，有必要申明这一点，字典字段绝不能只存文本，这关系到整个项目的统一性协调性完整性。

在前面的“设计规范”——“基本原则”——“连接查询”部分专门提到过字典字段，有时候，在较为核心的业务表中可能出现五个甚至更多的字典字段，比如订单表，如果只存字典编码，如何一次查询出核心业务表中字典字段对应的文本信息？全部都使用对字典表连接查询的话效率太低了，这种情况下，通常将字典编码和对应文本一并存入，至今未找到既可以只存字典编码又可高效查询的方法。

在程序开发过程中，具体到某一常用字典类型、字典项的编码和文本信息需要即时获取，以往的习惯在程序中建立一个常量类，所有用到的字典数据在里面标明。前端部分也会有一个常量JavaScript文件，和后台程序中的常量类功能一样，记录常用字典类型、字典项编码信息。比如：

public static final Integer DICTORY\_ITEM\_ID\_USERTYPE\_PERSONAL = 1;

public static final Integer DICTORY\_ITEM\_ID\_USERTYPE\_ENTERPRISE = 2;

public static final String DICTORY\_ITEM\_TEXT\_USERTYPE\_PERSONAL = "个人用户";

public static final String DICTORY\_ITEM\_TEXT\_USERTYPE\_ENTERPRISE = "企业用户";

JavaScript文件：

var DICTORY\_ITEM\_ID\_USERTYPE\_PERSONAL = 1;

var DICTORY\_ITEM\_ID\_USERTYPE\_ENTERPRISE = 2;

var DICTORY\_ITEM\_TEXT\_USERTYPE\_PERSONAL = "个人用户";

var DICTORY\_ITEM\_TEXT\_USERTYPE\_ENTERPRISE = "企业用户";

这里单独强调一点，目前使用的region地区字典表，存放全国省市县的信息，里面有必要加入一个是否可见字段（is\_display ）和一个备注（remark ）字段。另外，通用地区字典表应该考虑到世界各国家、镇、乡、县级市、市下面区、市下面县的问题，中国一级行政区使用国家统计局定义的编码，世界国家使用ISO-3166-1标准中的Alpha3编码。当前的设计在存储、处理世界上国家的编码、中英文名称等信息时已经不适，而要新增对镇、乡、县级市、市下面区、市下面县的存储，编码规则也要重新整理才可以满足，届时可以参考网络中的标准数据，不要随意编制。

相信以上关于字典的处理还有更好的方法，在后续的开发中会探索留意总结。

B.系统日志表

以往设计的项目中日志分三部分：一部分是系统登录日志，一部分是用户操作日志，一部分是运行异常日志。如果系统上线前测试没问题，运行时异常出现的概率较小，一旦出现多是外部因素所致，比如网络中断数据库失联等等，这部分通常记录在本地日志文件中的。系统登录和用户操作则要记录在数据库中，要有相应表存储，下面给出之前设计的表结构：

/\*==============================================================\*/

/\* Table: LoginLog \*/

/\*==============================================================\*/

CREATE TABLE [dbo].[LoginLog](

[loginLogId] [varchar](50) NOT NULL,

[userId] [varchar](50) NULL,

[loginTime] [datetime] NULL,

[logoutTime] [datetime] NULL,

[durationTime] [bigint] NULL,

[ip] [varchar](50) NULL,

[status] [varchar](5) NULL,

[browser] [varchar](50) NULL,

[resolution] [varchar](50) NULL,

[ipAddress] [varchar](50) NULL,

[operatingSystem] [varchar](50) NULL,

[userName] [varchar](50) NULL,

[userPassword] [varchar](50) NULL

)

/\*==============================================================\*/

/\* Table: OperationLog \*/

/\*==============================================================\*/

CREATE TABLE [dbo].[OperationLog](

[operationLogId] [varchar](50) NOT NULL,

[loginLogId] [varchar](50) NULL,

[userId] [varchar](50) NULL,

[operationTypeCode] [varchar](2) NULL,

[operationTable] [varchar](50) NULL,

[operationContent] [varchar](500) NULL,

[isSuccessId] [varchar](2) NULL,

[resultCode] [int] NULL,

[objectRecordId] [varchar](50) NULL,

[resultMessage] [varchar](1000) NULL,

[totalCount] [int] NULL,

[addTime] [datetime] NULL

)

过去的设计中，几乎所有表中都设有create\_time、create\_user、update\_time、update\_user四个字段，如果操作日志表设计的合理、又有相应程序配合足够严密的话，除核心业务表外这四个字段完全可以拿掉， 所有针对数据库的增删改查动作都会在日志表中有迹可查。

但现在的系统登录日志及操作日志部分仍有很多问题未处理好。

第一个是登录日志中对于登录时长的记录，因为目前写的程序无法确定用户离线时间，所以这个字段的数据无法计算。

第二个是操作日志中对非登录用户操作的记录。接触的大都是Web业务系统，非登录用户是不能对系统进行操作的，但如果项目中有对外部分（Public）呢，比如网站，那这些查询动作要不要记录呢？非登录用户的查询动作记录的意义不大，除非是搜索动作——如果有这个功能的话，有注于分析用户习惯。但如果业务系统较大，搜索部分比较重要，应该有单独的搜索日志记录功能才好。

第三个是操作日志中对于增删改查动作的细分。查询操作较为频繁，无论用户是否登录，其在系统中的每个动作都有可能触发对数据库的查询，对这部分的记录处理的不是很好。不是不好记录，而是不好对查询记录进行分类。通常情况下，增删改这类写操作都要记录日志的，不过有些查询操作却没有记录日志，比如当用户在登录系统时，也是调用的查询方法，但有登录日志功能专门对这个操作进行记录，那这个查询操作实际并无必要记录在操作日志中。再比如，用户在新增卡片信息时，系统可能要先判断一下这个卡号是否已经存在于数据库中，并给出相应的提示，那这个系统业务自身执行的操作还要不要记录在操作日志中呢？过去没有对这些查询进行记录，但是不是可以完整记录，然后对操作记录进行分类，哪些是用户直接解触发的、哪些是间接触发的由系统业务执行？可不可以把操作来自于哪一个页面哪一个控件、执行的哪一个动作都进行分类记录，以方便后期查阅呢？

过去接手的项目，未登录用户多是没有写权限的。现在的程序架构，日志记录功能放在了Service层，如果对查询动作细分其来源的话，则当前架构需要调整。如果仅仅区分是否为登录用户的操作还好，如果要记录哪些是用户直接操作、哪些是由系统业务执行、甚至细分动作来源就不容易了，这些都是要做参数由Controller传入Service的，或者将日志记录由Service中去除，分散到Controller中，真这样做内聚性就差了。目前的架构，可做的就是，对于未登录用户的操作及用户间接触发的动作不做记录，只记录登录用户直接触发的动作。

第四个是操作日志中对于增删改查详细内容的记录。比如增加的话增加了什么、查询的话查到了什么、删除的话删除了什么，修改的话修改了哪些属性、修改前后属性值各是什么。把这些操作结果序列化成某种格式存储在日志表中到也不难，问题是都存入数据量太大了，想不到可以精确存入少量数据的方法。

第五个是操作日志记录的备份。当前阶段的数据库中，操作日志表是数据量增长最块的，过去并无有效方法处理，只能用脚本定期删除历史数据。网上有建议，日志类数据不要直接存储在MySQL上，优先考虑Hbase或OB，如需要存储请找DBA评估使用压缩表存储。也有建议，采用活动库与历史库分开设计，提高系统访问即时数据的效率。认为合理又相对简单的办法还是后者，定期迁移过期日志数据到备份库中。此过程应该也可以用脚本处理，或者尝试寻找用工具自动处理的方法。

对于日志记录存取相关的问题，都未能找到特别好的方法处理，以上解决方案只是建议，也会在后续的开发中继续探索留意总结。

C.用户权限表

对于权限的控制，由高到底大致分这几个级别：系统级、模块级、菜单级、页面级、控件级、数据级。菜单、页面、控件的落脚点都在请求链接地址上，数据级的权限最难控制，不同的角色对不同的数据有不同的操作权限，要对数据进行分类，自己过去负责的、接触的项目中很少有对数据级权限做到灵活控制的。系统级的权限表示可以控制不同项目，平时的单一项目无须考虑。就是说在现阶段只要控制好中间几项就可以了，相关设计都是可复用的。

如果对用户、角色、权限相关的内容详述的话，再写这样一份文档也讲不完，这里只简单一提，具体设计不讲。业务系统的权限控制功能必不可少，但真正设计好的却不多，这部分的根本解决方案还是要开发通用的权限管理系统，对所有项目统一管理。对于软件研发公司、团队，这是一劳永逸的工作，但必须要有足够经验能力的人牵头做。

D.并发编辑处理

在设计程序架构时应该考虑这样的问题，比如当一条会诊记录的状态是刚提交时，那么会诊申请医院和会诊接收医院都可以对这条记录执行操作，会诊申请医院可以取消修改会诊信息、会诊接收医院可以拒绝或安排专家。如果两个用户在同一时间段操作的话，很可能出现冲突。比如会诊申请医院有可能正在取消页面中填写取消会诊的原因，而会诊接收医院恰好也进入了会诊处理页面正给同一会诊申请记录安排专家。类似的情况还很多，比如当系统用户位于修改页面中正在编辑某条数据时，恰巧有另外的用户对此条数据执行了删除动作，这时修改完后的提交动作就会出现异常，如何避免这一类的问题？

有个想法是，可以给数据库表加个布尔型的锁字段，当有用户可能对此表中的某一记录执行写动作时，比如进入修改页面，标识锁字段（加锁），任何用户在想对此条记录执行其它的修改动作前，要先查看锁字段是否处于加锁定状态。可问题也有，BS的系统中程序如何判断用户已经停止可能的更改动作，而恢复此标志为解锁状态呢？比如用户可能进入修改页面后，并未执行修改动作，而是直接在浏览器中关闭了此页面，但此时记录却还处于被锁状态。

此种类似悲观锁的处理方式之所以不可行，因为以上设计思路中妄想用数据库中的表字段标识用户进出页面的动作，可在BS的系统中，程序是无法监听到客户端页面释放的。再有，此设计中锁的资源其实质是数据库中的表记录而非页面，而在实现时却想用其去锁页面，也不甚合理。由此，对于锁，首先应该明确锁的内容，其次应该明确添加、释放动作是否可被捕获。

那换种思路，不修改数据库表设计，不加额外的锁字段。后台程序在执行修改动作时先做判断，注意此处说的是后台程序而不是用户，是在后台程序执行针对数据库的写操作时而非用户进入修改页面时。先根据主键对相应记录执行一次查询，如果有早前的操作导致本次操作无效，则给出提示，比如该记录已被删除、该会诊申请已被取消不能安排专家等等。但此种解决方案只能解决记录被删除、会诊单状态被修改这种特定性的情况，无法处理该记录和提交修改前的初始记录不同的广泛型问题，因为程序不好在执行写操作前，逐个判断整条记录的每个具体字段。有可能此条记录已被删除、有可能是会诊单状态有更改、也有可能是病情描述有修改。如果申请医院修改的是病情描述，那接受会诊医院再做写操作提交，就会把之前的病情描述覆盖掉。

所以对于并发编辑的问题，目前唯一可行的方法，还是在数据库中加入一个标识字段。不过此字段不能是布尔型，而是数值型或日期时间型。程序在执行修改前先执行一次查询，检查此字段当前的值（时间）和本修改提交前获取的原始值（时间）是否一致。如果一致，则此字段的数值发生变化，同时正式执行针对记录的修改动作；如果不一致，说明此条记录在当前的修改期间被其他人修改过，此修改动作作废，给出前端错误提示。但这样的不好之处是，用户有可能在操作完之后才发现，自己的之前的操作是无效的。比如在修改页面都已经填写完要修改的信息了，点击提交才给提示此条记录已被删除。

理想的悲观锁解决方案应该是：既然有人正在编辑此条记录，那在这段时间，其他人则不能进入针对这条记录写动作的页面、不能执行针对这条记录的写动作。但前面也说过了，在BS的程序中，前端页面的释放动作无法捕获，注定此方案不好实现。

严格来讲，所有数据可修改的表中都应该有这样的锁字段来处理并发，可以在程序的业务逻辑层对修改方法进行统一控制，校对锁字段。以往的数据库设计、程序设计中，对并发编辑的处理考虑欠缺，很多处理也不甚得当，要引起注意。

E.历史版本设计

在数据库设计过程中，经常会遇到一个需求，就是希望把操作之前的数据保留下来，能够看到操作之前是什么数据，操作之后是什么数据。对于这种需求，我们可以使用保留历史数据或者使用版本来实现。

版本号是一种常见的版本设计方案，就是在要进行历史数据保留的表上面增加一个版本号字段，该字段可以是datetime类型，也可以是int类型，每进行数据操作时，都是创建一个新的版本，版本是只增不减的，所以只需要拿到最大一个版本号，就能得到最新的业务数据。

版本号和上面针对并发编辑加锁的解决方案类似，两者表现在在数据库中的设计完全一样，都是加一个标识字段，可以是数值类型或时间类型。不过程序的处理不一样：版本号的设计中程序会在同一张表中存下记录修改的历史；标识锁的设计，只是为了给某一记录加锁，处理并发编辑问题。

个人不建议将修改的历史记录和业务数据混淆在一起存放于业务表中，因为程序读取数据时要先对版本号字段进行判断、程序的修改动作在程序中实质也变成了插入动作，多了很多逻辑处理，况且一起存放容易使业务数据表的数据量因历史版本记录而膨胀。

历史版本记录可以单独建历史表存放，其实就是建立相同Schema的表（当然也可以添加更多的字段用于记录额外的历史版本信息），该表只保留历史版本的数据。这有点像一个归档逻辑，所有历史版本我们认为都应该是不经常访问的，可以扔到单独的表，对于现有生效的版本，仍然保留在原表中，如果需要查询历史版本，那么就从历史表中查询。使用历史表记录历史版本，就是在程序对数据进行增删改操作时，先在历史表中留痕。

对于会诊单这种核心的业务表，保存历史修改记录是很有必要的，但并非所有的表都一定要保存历史记录，对于其它一些非核心表，可根据实际情况斟酌是否建立对应历史表。不过合理的设计应该是一套完整的体系，没有重要不重要之分，只要存在的都是重要的，都要用严格统一的设计方案。为此或可以创建额外的历史版数据库，里面单独保存所有的历史表。

这里有必要再次提下前面系统日志表设计部分的介绍，里面说可以把每次的写操作序列化成某种格式存储在操作日志表中，如果操作日志的设计可以记录的足够详细灵活——增加的话增加了什么、删除的话删除了什么、修改的话修改了哪些属性、修改前后属性值各是什么，具体到某张表的历史记录或可不用再单独处理。

以往的数据库设计中，多没有考虑详细的历史记录功能，后面应该注意，再仔细考虑下操作日志（OperationLog）表的设计和存储（优先考虑Hbase或OB），看看可否将其设计的足够详细灵活。

F.大数据可视化

在“设计规范”——“基本原则”——“连接查询”部分提到存有大量可视化图形展现的系统，对于这种系统如何提取通用部分，以便高效、严密、灵活而又统一的控制，尚未有好的解决方案。业务、约束在这样的系统中不是核心；前台只有读而没有写，对读的要求很高，对写却不在意；只要统计分析的结果，而不要源数据。按理没有源数据，自然没有统计分析，但有时这是个大数据平台，这些源数据根本拿不到，它来自于整个市场、整个互联网。但是统计分析的结果可以从网上搜到相关，跨过源数据而直接显示。

对市场和行业进行大数据分析是非常困难的，不是说分析困难，而是拿到数据困难，单凭爬虫团队从网上爬取的那点乱七八糟的数据，用来分析根本不现实。互联网之外，行业内部的数据又都掌握在政府或企业手里，除非有非常亲密的合作关系，这些数据也是拿不到的。而且手里有大数据的机构或企业，往往自己就有能力进行数据分析，有大数据却没能力分析又找到我们做，是最理想而少见的情况，这就无需担心数据源，项目也就是纯粹的技术问题了。

做大数据分析平台，当然首先得有大数据，否则出来的系统就是无源之水、无本之木。问题是现在没有，还想做数据分析可视化展现，该当如何？

首先是需求提出者那边要立足于实际，不要凭空的去想要一个东西。没有数据也不知道数据具体从何而来，却妄想做大数据分析，这是一切的症结所在，所接触项目的痛点根源就在这里。但另一方面我们可控制的又很少，有人愿意付钱，无论痛不痛到你这里都要做的，没得选择。

其次是产品经理那边，在接到客户的模糊需求后开始设计产品，先考虑的问题是实现，要设计这个功能数据源从哪里来，可否实现？这些客户那边不考虑没办法，产品这边如果也不考虑而天马行空的设计，最后到开发那边就全懵逼了。如果没有源数据，那要展示的统计分析结果，互联网上可否获取明确的数据？这里说考虑实现不是泛泛的，而是对于设计的系统中的每一处，所需的数据都要有明确的数据来源方可。

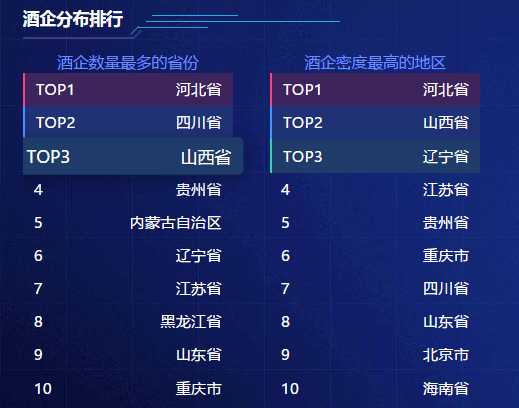
按常理，产品经理在设计前应该先过项目经理这边，详细沟通之后再着手设计，而且设计过程要和项目经理实时沟通。但当前所在公司的情况就是产品经理在拿到需求后直接设计，很容易脱离于业务，且其没有能力考虑到后期具体的实现问题。如果是纯粹的后台业务系统，交给产品经理设计问题会更多。等到完成设计再交付到项目经理这边组织人员开发，有不合理的地方项目经理很难再强力左右，总不能让他再重新去设计，只能默许当前的设计、提些细微处的修改意见后着手开发。再说，工作流程有问题，前期及设计过程中不和项目经理实时碰面，返回重新设计多少次还是会有问题。这样设计和实现就脱节了，一个烂的系统在前期就已经注定。

最后到开发这边，应该先确定系统涉及哪些业务、自己可以拿到哪些真实的源数据、哪些真实的数据处理析结果，对于有源数据的部分可以先理清其中的关系，对其进行关系化存储，而对于没有源数据只有数据处理分析结果的部分则要另当处理。

自己以往做的这种项目，都是先理清业务关系，然后结构化设计数据库，如果最终只有数据处理分析结果而没有源数据，则制造模拟源数据，以这些模拟的源数据计算出准确的分析结果为最终目标。在制造模拟源数据时非常不便，因为结构化的存储、业务关系，表与表间互相干涉，影响着SQL查询出的最终分析结果。

也就是在这时开始想，对于这种前台界面中只展示数据处理分析结果的项目，结构化的数据、关系型数据库可能不是最好的选择，用非关系型数据库会不会更灵活一点？

只有分析结果或只有源数据都还好说，问题是大部分项目都介入两者间，有部分有源数据，有部分只有分析结果没有源数据，这些有的源数据和有的分析结果间可能还有些业务上的联系。比如酒企信息这些源数据本可以获取一部分的，但各省份的所有酒企的具体信息却是拿不到的，但同时各省份拥有的酒企数量排名这个统计分析结果却是从网上可以搜到的。这TM就尴尬了~~~本来在关系型数据库中有酒企和地区表，多对一，但为了得到这个分析结果，我要在酒企表中制造大量模拟数据，来平衡得到下面的分析结果。如果不这样做，除非针对这种只有分析结果没有源数据的部分单独设计表存储，但这样关系型数据库就全乱了，也发挥不了他应有的作用。



现在想到的一个办法是针对这种只有分析结果没有源数据的部分单独存储成XML数据，或者就是用非关系数据库存储，对那些有源数据而又涉及业务的部分还是用关系型数据库存储，在程序中对这两种数据分别处理。以往没有试过这种方法，后面可以尝试下，是否会让后期的开发及数据维护更简单些？

自己后期这种纠结的根源，还是在前面的需求提出和产品设计中，如果这两个步骤自己可以强力左右，定然不会让后期开发出现这种进退不得的情况。过去一些项目，项目经理、产品经理、技术经理的角色都是一人独揽，反而是效率最高、对项目最为有益的。个人是敏捷开发的极力倡导者，在项目调研、设计、开发、实施的整套流程中主张独裁，极其厌恶带庞大臃肿的团队、极其厌恶使用低效的传统开发模式。

G.其它注意事项

在建表时的字段排列顺序也要注意，虽然在使用相应的数据库管理工具时，一般可以按名称对表字段进行排序显示，但建表时默认排序最好和字段的重要程度有一定关联，这样开发人员在查看表结构时会一目了然。而且表一旦建完，在一些数据库中想要再修改其字段默认顺序就不容易了。一般是基本的、核心的字段在前，次要的、扩展的字段在后。这个默认的顺序，约是你在建本张表时，脑海中先后浮现出的字段的顺序。

在数据库表设计过程中，不建议提前加扩展字段，当前想到有几个字段则就有几个。加扩展字段主要是为了方便后面开发过程中需要新增字段的情况，但是，最后需要增加的字段可能和你设定的字段的类型不一样，这样扩展字段还是没用的，需要手动更改，再说字段名称也都要改，那还不如不加扩展，有需要时再添加呢。

设计完成后可以用工具添加些测试数据，不过工具添加的数据多不规则，这个根据情况而定吧。

**9、梳理总结**

“设计规范”几乎把数据库设计过程中的各个注意点都提到了，且详细讲述了各注意点涉及的知识面，并将自己的设计理念灌入其中。现把此部分的要点梳理如下：

a． 数据库设计过程中使用PD工具。

b． 百张表内的数据库由一人来设计。

c． DBA应该是在被需要时出现，而不应该被强制需要。

d． 开发过程中的规范和约定采用大一统的方式，严格限制脱离中央管控的脚本或代码出现。

e． 设计原则是尽可能让粒度小、容忍度高。

f． 负责数据库设计工作的人应该是最懂项目、最懂业务需求、最有设计经验的人

g． 把项目相应部分的决定权交到相应最懂的人手中，其他人不要干涉。

h． 这里建议将MySQL数据库编码设置为utf8， SQLServer、Oralce可先采用默认设置，有需要则根据实际情况做相应变动。

i． 禁止使用存储过程、触发器、函数、视图、事件等高级功能。

j． 数据库设计尽可能的遵循关系数据库范式。

k． 设计过程中要考虑尽可能减少后面程序编写时查询的复杂度

l． 每张表必须有只用来标识唯一的主键，主键为UUID，统一命名为id，统一为char(32)类型。

m． 现阶段暂时不考虑逻辑删除，仍旧使用物理删除。

n． 对于中小型项目，数据库设计过程中无需过多的考虑性能、安全的问题。

o． 不同的关系型数据库在字段类型的具体化上差异较多，但具体化的字段类型再多，无外乎这几种：字符、数字、日期、二进制。

p． 表中应该尽可能避免可为NULL的列，且尽可能显示设置默认值，尤其是被索引的列。

q． 如果数据库中某个字段有默认值，在程序开发过程中，对应实体类的属性应该设置同样的初始化值才合理。

r． 禁止使用float、double类型，建议使用decimal替代。

s． float：浮点型，含字节数为4，32bit，数值范围为-3.4E38~3.4E38（7个有效位）；double：双精度实型，含字节数为8，64bit数值范围-1.7E308~1.7E308（15个有效位）；decimal：数字型，128bit，不存在精度损失，常用于银行帐目计算。（28个有效位）

t． decimal(a,b) ，a指定指定小数点左边和右边可以存储的十进制数字的最大个数，b指定小数点右边可以存储的十进制数字的最大个数。最大精度38。

u． MySQL的decimal对应于Oracle的number，数据类型的精度：number(p,s) ，p： 1---38， s：-84---127。s代表的是小数位数，p代表的是总位数（整数位数和小数位数）。

v． 数据库Oracle服务端的exp.exe工具远程导出源数据库，导出导入命令：

cd D:\app\bsc\product\12.1.0\dbhome\_1\BIN\

exp origin\_username/origin\_password@origin\_tnsname file=C:\Users\Administrator\Desktop\20150508.dmp owner=origin\_username

imp target\_username/target\_password@target\_tnsname file=C:\Users\Administrator\Desktop\20150508.dmp ignore=y full=y statistics=none

w． 要求日期时间类型的字段，尽可能精确到时分秒，用datetime类型。

x． 字典编码字段，之前在SQLServer中设计数据库时统一使用char(2)类型，Oracle数据库中统一使用number(2)，在MySQL中统一使用tinyint(2)。

y． 统一命名为remark，字段类型为varchar(200)，最多100个中文字符。

z． 推荐统一使用int(10)做为所有表中的排序字段类型。

aa． 对于表的注释，要求简单明了，先说明表中文名称，句号分隔，然后跟功能说明，无需特别说明的，则只说明中文名称即可。同时，为了最终生成代码注释的可读性，不允许在说明中出现“表”字眼。

bb． 主键字段，无须特别注释，注释就两个字：主键。

cc． 外键字段注释：

` applicant\_doctor\_id` char(32) DEFAULT NULL COMMENT '申请医生。外键，引用医生表（doctor）的主键（id）。理论上可填多个，以“|”分隔'。

` applicant\_doctor\_name` char(32) DEFAULT NULL COMMENT '申请医生。名称，和医生表（doctor）的医生名称字段（name）对应。理论上可填多个，以“|”分隔'。

dd． 字典字段注释：

`status\_code` varchar(5) DEFAULT NULL COMMENT '转诊单状态。编码，对应数据字典表（dictionary）中的编码字段（code）。目前先定义6个状态：01 已提交，02 已取消，03 已审请，04已安排，05 已拒绝，06 已结束。03是向HIS系统执行审请，04是HIS系统已安排入院日期，下转诊时这两个字段没有区别。'

`status\_text` varchar(50) DEFAULT NULL COMMENT '转诊单状态。文本，对应数据字典表（dictionary）中的字典项名称字段（name）。目前先定义6个状态：01 已提交，02 已取消，03 已审请，04已安排，05 已拒绝，06 已结束。03是向HIS系统执行审请，04是HIS系统已安排入院日期，下转诊时这两个字段没有区别。'

ee． 普通字段注释：

`bed\_id` varchar(50) DEFAULT NULL COMMENT '入院床位号。HIS系统安排并反馈的入院床位号，如果是下转诊，此部分由医生手动填写'。

ff． 五种约束类型：唯一性和主键约束、外键约束、检查约束、空值约束、默认值约束，对应五大关键词，UNIQUE和Primary Key, Foreign Key, CHECK, NOT NULL, DEFAULT。本规范中要求，除检查约束外，其余四种都要在数据库中加强约束，同时，程序也会根据数据库中约束的设置做相应处理。

gg． 当数据量达到一定程度，仅通过程序优化对性能提升有限时，可以考虑同时对数据库进行调校，这时会优先考虑建立索引。提到的几个注意点：索引外键、索引经常GROUP BY、ORDER BY的字段、不要索引blob/text字段、不要索引小型表、索引多了影响插入删除速度、在条件表达式中经常用到的不同值较多的列上建立检索。

hh． 对于并发编辑目前唯一可行的方法，还是在数据库中加入一个锁标识字段。

ii． 优化操作日志（OperationLog）表的设计和存储（优先考虑Hbase或OB），来代替单独历史表的设计。

jj． 大数据分析可视化平台，针对只有分析结果没有源数据的部分可尝试单独存储成XML数据，或者就是用非关系数据库存储。

kk． 在建表时的字段默认排列顺序也要注意。

**三 命名规范**

**1、引言**

在前面提到过，数据库设计过程中表、字段等的命名规范也算是设计规范的一部分，但因为牵涉的内容较多，所以将其单独列出。不过设计规范更多的是为了确保数据库设计的合理性、为了项目最终的协调稳定性，而命名规范则更多的是为了确保设计的正式和统一。公正的讲，数据库中表字段等等以什么样的方式命名、取具体什么名字，并不会直接影响到项目的稳定性，不是说叫黑猫项目就是正常的，叫白猫就运行异常了。

制定规范的直接目的是约束设计行为，最终目的是确保设计的合理统一。规范虽然是有丰富项目经验的人制定的，但维护的却不是某个人的意志，而是项目的意志，因为遵守此规范对项目是好的有利的，此规范才有意义。所以规范是为了项目利益最大化而在团队人员中形成的一种约定（貌似约定的英文单词Convention本身就有规范的意思），所有参与设计的人员都要遵守此约定，所有参与开发的人员都会依此约定解读设计。我们约定，所有的主键统一命名为id，结果有设计人员违反约定将一个非主键字段命名为id，约定被打破，共识也就被打破，设计人员之间、开发人员与设计人员之间的沟通就出现了隔阂。

设计规范更多的是为了合理，命名规范更多的是为了统一，团队协作中，统一在某种程度上比局部设计开发的好坏更重要。违反了约定，局部设计开发的再好，反而可能影响到项目整体的稳定协调。

约定优先于配置（Convention Over Configuration）。

在“设计规范”中提到过一些命名规范，也详细讲述了表、字段的类型、注释等属性的设置，为什么要求主键统一命名为id、统一为char(32)类型，为什么要求浮点型数值统一为decimal类型？我们希望团队中所有人看到设计成果，一眼就可以明白这个字段是做什么的、代表的含义是什么，可以但不止于见名知意。再者，当前的开发模式，前后端代码及数据库文档、程序文档、接口文档等等大都是由工具生成，而其最底层的依据就是数据库，表、字段的命名注释同时会影响到工具生成的文档、代码中的类属性方法甚至是前台页面的命名注释，数据库设计命名的规范关系到整个项目的规范。

命名规范会分四个大模块来介绍：基本规范、名大小写、具体规范、特别说明，各大模块下面有的会有子模块特别说明。

**2、基本规范**

A.可用字符

数据库、表、字段等所有名称的可用字符范围为：A-Z，a-z， 0-9 和\_下划线，除此外不允许使用其它字符作为名称。数据库及表名均不允许出现数字，字段名除非特殊情况不允许出现数字。

在前面介绍关系范式时曾提到过一个破坏范式的例子：平时的多图片上传功能，可能只设计一个字段存储图片名称，这样字段值中就会包含多个图片的名称，里面用|或其它符号分隔。像这种情况，其实也可以设计成三五个字段image\_name1、image\_name2、image\_name3……分别存储，然后限制可上传图片个数，这就是字段名中可出现数字的特殊情况——虽然也不建议这样设计或取名。

B.命名方式

数据库、表、字段等所有名称使用英文单词或英文短语或相应缩写，禁止使用汉语拼音，且均使用单数名，例如：对存储客人信息的表命名为customer而不是customers。名称应该清晰明了，能够准确表达事物的含义，遵循见名知意的原则。

Oracle表、字段等名称统一使用大写，单词间用\_下划线分隔；SQLServer数据库、表等名称采用Pascal命名法，字段名称采用Camel命名法，大小写字母混排；MySQL数据库、表、字段等名称统一使用小写，单词间用\_下划线分隔。至于为何这样规定，下一个模块会有详细介绍。

Oracle相对特殊，通常的操作顺序是，先创建数据库实例，然后创建表空间，然后创建用户并设定此用户的默认表空间，最后在此用户下建表。多数情况下我们都是只建一个实例，然后在此实例下建不同的表空间、不同的用户，根据不同的用户来区分不同的库。关于实例、表空间及用户的命名方式并无限制，可以采用大小写混排，也可以只用大写或小写，但对于表和字段，我们要求统一为大写。

我们要求统一为大写或小写的名称，两个单词间用\_下划线分隔，SQLServer使用Pascal或Camel方式命名。这些不仅仅是为了数据库设计的可读性，也是为了最终生成代码的可读性。这里简单介绍下编程中常用的三种类、变量、函数等的命名方式：

a． 匈牙利命名法。由微软的一位匈牙利程序员Charles Simonyi 提出，相对复杂，首字母小写，基本原则是：变量名＝属性＋类型＋对象描述，其中每一对象的名称都要求有明确含义，可以取对象名字全称或名字的一部分。匈牙利命名法主要在C或C++这种面向过程的程序语言中使用，如果用在Java、C#这种面向过程的语言中就很别扭。

不过自己在写Web前端页面或脚本时，借用了这种命名方式，form表单中涉及的常用HTML标签不外乎如下几种：label、text、button、submit、password、textarea、radio、checkbox、select等，那我在给表单元素命名或者说是给id或name赋值时，就会将元素类型做前缀，例如用户名输入框为textName、性别单选按钮名为radioGender。这样做的好处是我在编写脚本时，根据id或name名称一眼就可以看出这个表单元素是什么类型。在修改页面中初始化表单数据时我可以直接遍历表单元素、根据元素名称判断出元素的类型进而采用适当的赋值动作，而不用逐个选择元素去赋值。

在ASP.NET编程中，如果使用微软的服务器控件，在命名时我会用控件类型做名称后缀，例如Name\_TextBox、Gender\_RadioButtonList等。之所以不再将类型做前缀，一来是VisualStudio本身默认的服务器控件命名方式即时如此，控件类型做后缀；二来是因为服务器控件的类型名称太长，而自己又不愿用缩写，因为没必要，VisualStudio的提示功能强大，后缀的长度不会影响到编程速度。

b． Camel命名法。即骆驼式命名法，首字母小写，采用该命名法的名称看起来就像骆驼的驼峰一样高低起伏。Camel命名法有两种形式：

第一种是混合使用大小写字母，例如englishName、fartherCode。在Java中，属性名和方法名一般都采用这种命名方式，在C#中只有属性名采用这种命名方式，我们在前面也规定，SQLServer中字段的命名也采用这种方式。

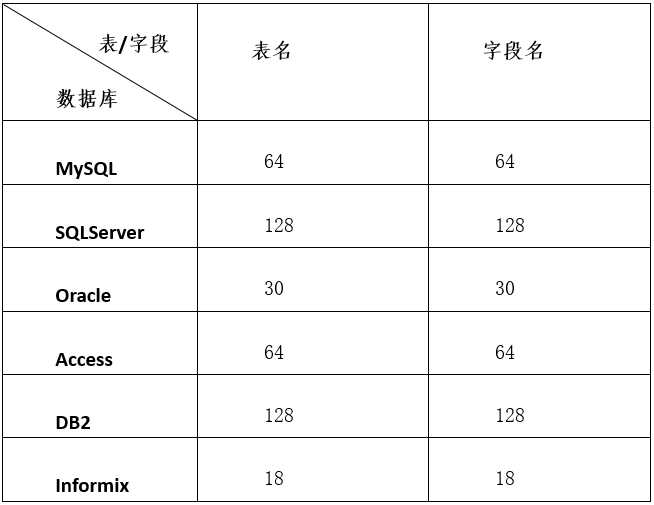
第二种是单词之间加下划线，例如english\_name、farther\_code。我们在前面规定，Oracel和MySQL表、字段的命名都采用这种方式，不过我们要求Oracle全部使用大写字母，MySQL全部使用小写字母。再者，无论是在Java还是C#，甚至是在JavaScript中，所有的常量，都使用这种命名方式，但和Oracle表字段的命名方式一样要全部使用大写字母，比如前面的设计规范中介绍数据字典表时，字典类型、字典项的编码和文本信息需要即时获取，以往的习惯在程序中建立一个常量类，所有用到的字典数据在里面用常量标明，这时常量的命名方式即是如此。

c． Pascal命名法。即帕斯卡命名法，与Camel命名法类似，不过是首字母大写。在C#中，类名和方法名一般采用这种命名方式，在Java中类名一般采用这种方式。在前面也规定，SQLServer中数据库、表的命名也采用这种方式。

除数据库的设计外，不同编程语言、前端HTML标签、JavaScript脚本、样式等等部分都会涉及命名的问题，如果细细整理，项目开发中每个子模块的命名规范都够再出一份长篇文档的。这里只简单介绍下三种常用的命名方式，其它部分的命名方式只是一提，重点还是在数据库的命名规范上。前面说过多次，程序、文档甚至前端页面有大部分通过工具自动生成，只有数据库严格按要求来命名，才能根据不同的编程语言编写不同的代码模板，统一控制生成部分各处的命名方式。比如，我们要求在MySQL数据库中，表名都使用小写，单词间用下划线分隔，交易记录表名称为trade\_log，那可以设定生成规则，对应生成的实体类名就是TradeLog，对应生成的Dal层就是TradeLogDal，对应的Service名就是TradeLogService，等等。可如果设计没有规范、不统一，那文档生成规则、代码生成规则、程序编写规则等等也就无法统一制定了。

C.长度限制

关于各种数据库管理系统（DBMS，Database Management System）本身对表、字段等名称的长度限制如下：



以上是从网络整理而来，Oracle、SQLServer及MySQL的限制长度亲自测试过。但也有说因为数据库和表的名字可能对应于目录和文件名，故而服务器运行的操作系统可能强加额外的限制。不过除了Oracle的限制长度过短外，其它的一般不会被超出。我们希望名称尽可能详细准确的表达事物含义，但如果过于冗长，就会给操作及后面的程序编写带来诸多不便。

D.单词缩写

自己以往设计数据库时，经常头疼于表、字段的命名，一来找不到好的单词去表述，二来有时可能涉及多个单词，导致名称过长。字段名过长带来的不便有限，最终影响的不过是程序实体类中的一个属性，可如果表名也过长，就比较麻烦了，生成的程序各层间针对此表的类名、变量名都可能受到影响，给后期的编写带来很大不便。使用单词缩写又拿不准，找不到合适的缩写方式。这里建议当表名超过15个字符、字段名超过20个字符时就应该尝试用单词缩写重新命名，如果名称长度在此之内，原则上讲则尽可能不用缩写以使表述具体清晰，表、字段最终的名称长度要严格控制在30个字符以内。关于单词缩写规则如下：

a． 如果可以在字典里找到一个词的缩写，就用这个作为缩写，比如：Monday=Mon、December=Dec ，可在此网站下查找到一些英文单词的缩写：<http://shortof.com/>；

b． 可以删除单词元音(词首字母除外)和每个单词的重复字母来缩写一个单词。比如：Current = Crnt、Address = Adr、Error = Err、Average = Avg；

c． 对于主从表，如果主表名称没有缩写而从表的名称需要缩写，则从表名称从第二个单词开始缩写，第一个名词尽可能和主表保持一致。比如企业基本信息表名称为enterprise，则企业诉讼表enterprise\_litigation可简写为enterprise\_ltg，企业证书表enterprise\_certificate可简写为enterprise\_crt。最终的数据库表及由数据库表生成的程序在集成开发环境（IDE，Integrated Development Environment ）中是按名称排列的，这样做是为了让相似功能的表、类文件排列在一起，方便开发者操作。

更详细的单词缩写规则介绍可以参考文档末尾的参考文献。

**3、名大小写**

理想情况下所有关系型数据库对于表名、字段名、字段内容等大小写的处理会有个大一统的方式，比如要求所有都是大小写敏感的，可实际的情况却是，不同的数据库及同一数据库在不同的操作系统下对大小写的处理都是不同的。以往笔记中记录的第一次遇到数据库处理大小写的问题是，做的一个登录页功能，测试人员发现输入用户名MengXianzhi或mengxianzhi均可以正常登录，但数据库用户表里只有一条用户名为MengXianzhi的记录，当时用的是SQLServer数据库。

A.编码和字符序的介绍

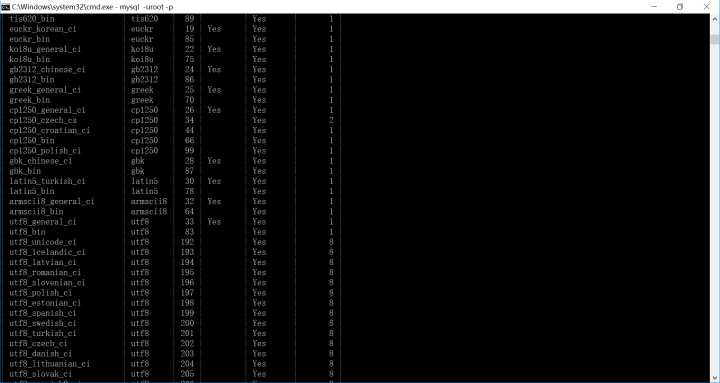
在前面介绍数据库编码时曾经提到，如果使用MySQL Workbench创建新的数据库，会要求选择Collation。Collation的字面意思是字符序，用于指定数据集如何排序、及字符串间的比对规则。可字符本来是不分大小的，这样对字符的>、=、< 操作就需要有个字符序的规则。Collation做的就是这个事情，你可以对表进行字符序的设置，也可以单独对某个字段进行字符序的设置，优先级从高到底可分为四种：服务器层、数据库层、表层、字段层，真正决定性因素是在字段层，如果没有指定则默认从上一层继承过来：字段层继承表层，表继承数据库层，数据库层继承服务器层，服务器层则需要设置，如果不设置默认为latin1\_general\_ci。

平时我们说的设置MySQL编码为gbk、gb2312、utf8或lantin等指的是字符编码，也就是Character Set。当表的Character Set 是lantin1时，若字段类型为nvarchar，则字段的字符集自动变为utf8。

可见数据库、表、字段的Character Set可逐级覆盖，这有点像上面说的四种字符序设置方式间的优先级关系。本规范中建议数据库统一设置编码为utf8，不仅仅是为了应付数据库间导入导出过程中、因编码格式不统一而导致的恼人的乱码问题，也是因为utf8是一种万国码（Unicode）。软件的国际化是大趋势 ， 而Unicode 是国际化最佳的选择。在MySQL中有两个支持 Unicode的Character Set：第一个是UCS2，使用 16 bits 来表示一个 Unicode字符；第二个是utf8，使用 1~3 bytes 来表示一个 Unicode字符。

那字符编码（Character Set）和字符序（Collation）之间的关系是什么呢？

每个Character Set 会对应一定数量的 Collation，在MySQL命令窗口中输入Show Collation;命令可以查看到所有字符序及其所属的字符编码列表：



同一个Character Set的不同 Collation 的区别在于排序、字符集对比的准确度以及性能，这里的准确度是指相同两个字符在不同国家语言中的排序规则可能是不同的 ，性能是指排序以及比对速度。例如：utf8\_general\_ci 在排序的准确度上要逊于 utf8\_unicode\_ci， 当然，对于英语用户应该没有什么区别，但性能上要略优于 utf8\_unicode\_ci，例如前者没有对德语中ß = ss的支持。而 utf8\_danish\_ci 相比 utf8\_unicode\_ci 增加了对丹麦语的特殊排序支持。

Collation名字的规则可以归纳为两类：<Character Set>\_<Language/Other>\_<CI/CS>以及<Character Set>\_Bin。CI 是Case Insensitive的缩写， CS是Case Sensitive的缩写，即指定数据库对大小写是否敏感。MySQL中Character Set对应的Collation多是CI的，CS这种校验字符已经逐渐被淘汰，gbk、gb2312、utf8等编码的所有Collation没有一个是CS的。Bin表示用二进制存储数据，用编码值进行比较，区分大小写。

在上面的截图中也可以看到，gb2312编码默认的Collation是gb2312\_chinese\_ci、gbk编码默认的Collation是gbk\_chinese\_ci、utf8编码默认的Collation是utf8\_general\_ci。按本文档中的规范，建议所有编码统一设置为utf8，如果不单独设置Collation，则按默认的utf8\_general\_ci，字段值是不区分大小写的。

那在字符序为CI的情况下，如何在执行SQL查询时区分字段值的大小写呢？假设用户表user中有两个用户：MengXianzhi和mengxianzhi，当我们执行如下查询时会得到两条记录：

select \* from user where user\_name = 'MengXianzhi';

如果要区分大小写，有下面两种方式可以精确查询：

select \* from user where binary user\_name = 'MengXianzhi';

select \* from user where user\_name = binary 'MengXianzhi';

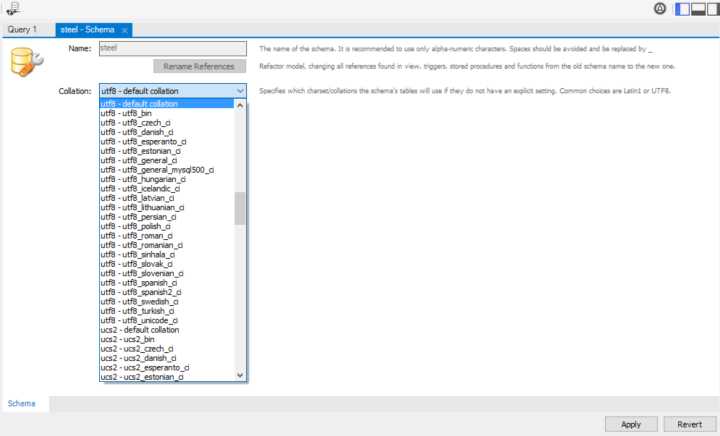
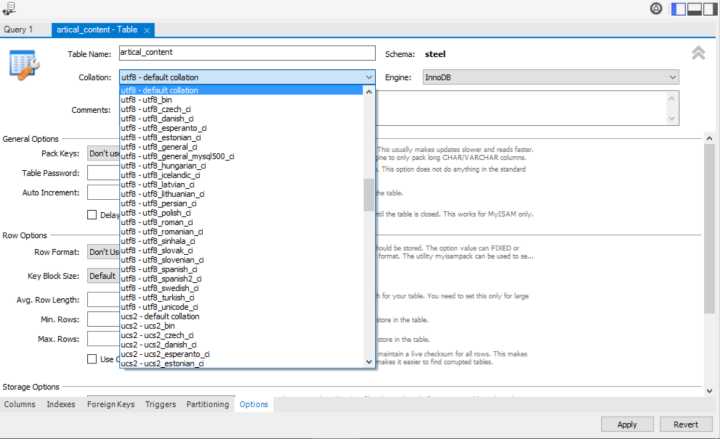
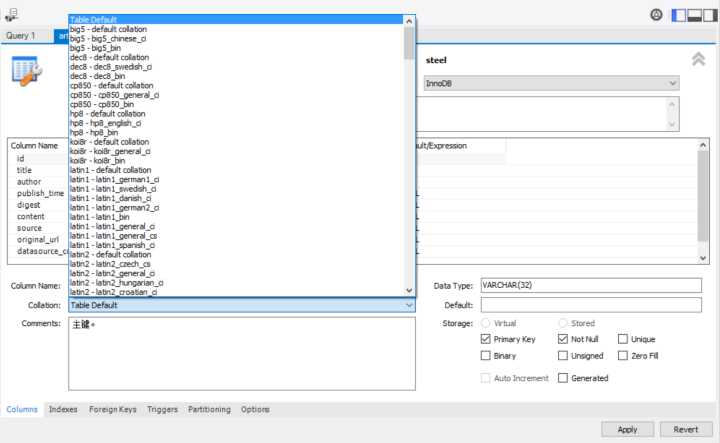
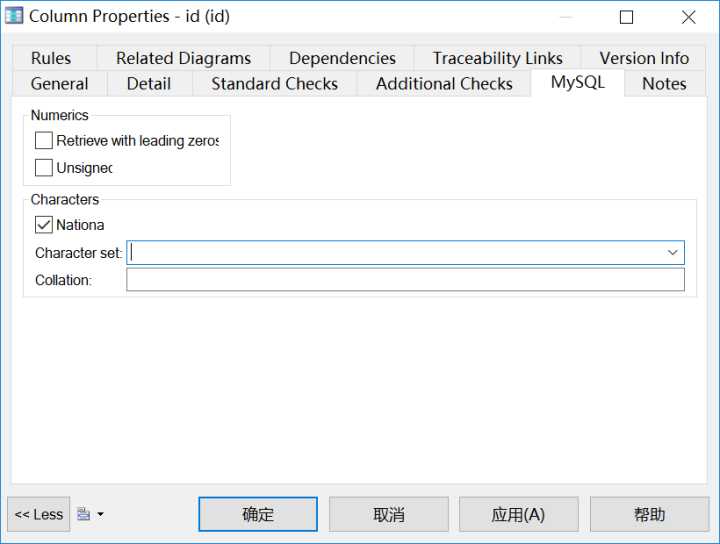
推荐使用第二种查询方式，这样可以保证当前字段的索引依然有效，而第一种会使索引失效。其实个人更倾向于建议统一设置数据库默认的Collation为utf8\_bin，也就是对大小写敏感。程序中针对数据库字段内容的比对查询处处都是，英文内容存储也处处都有，如果所有相关查询语句都加binary关键字，太过麻烦，不如在数据库中统一设置，这样也不会出现在本章节开头所描述的问题了。

如果不想在数据库中统一设置，也可以只针对表、字段单独设置，但非常不建议如此，因为这会导致局部配置和全局配置相悖 。一直坚持规范、约定、配置等尽可能采用大一统的方式，除非不得以。开放局部配置会导致配置的多样性，不利于统一管理维护，不过下面还是会简单介绍下局部配置的方法。

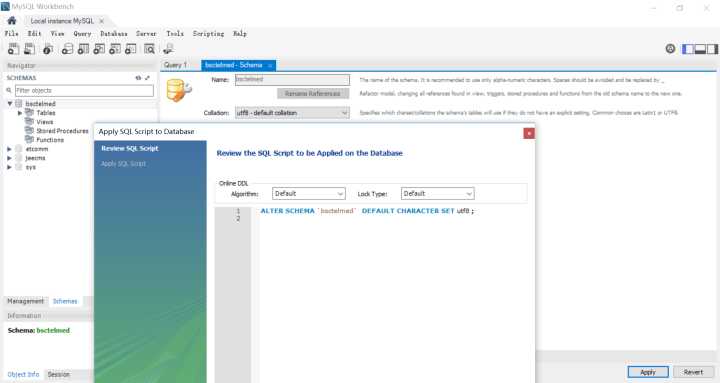
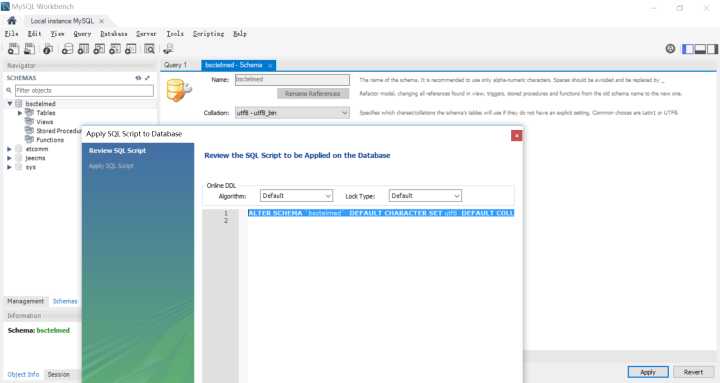
B.编码和字符序的设置

这部分会分别对比MySQL、SQLServer、Oracle三种关系型数据库的字符编码和字符序配置，先从MySQL开始。

在MySQL中，自己没有找到从服务器层面直接配置Collation的方法，但是数据库、表及具体字段设置Collation的方法都有。再就是在PD中未能找到全局设置Collation的方法，只找到了具体到某一字段设置的地方。截图如下，最后一张是PD中对某一具体字段进行配置的方法：

如果要直接更改某一个数据库的Character Set或Collation可以在MySQL Workbench中做如下设置：

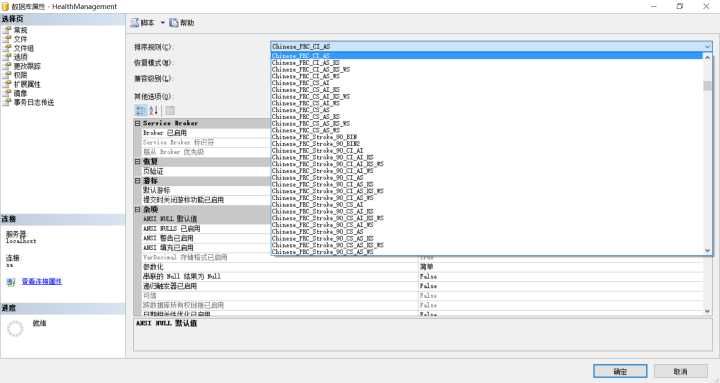
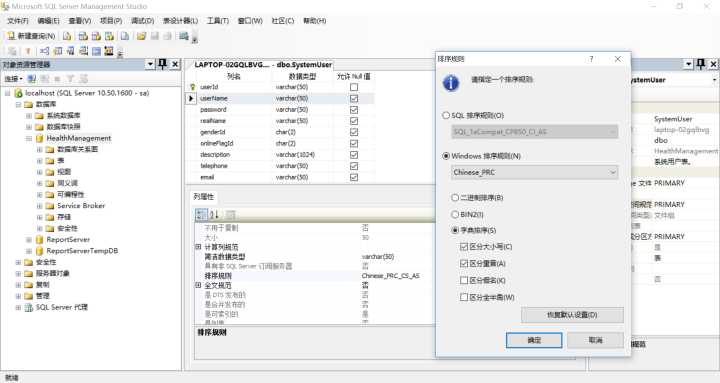
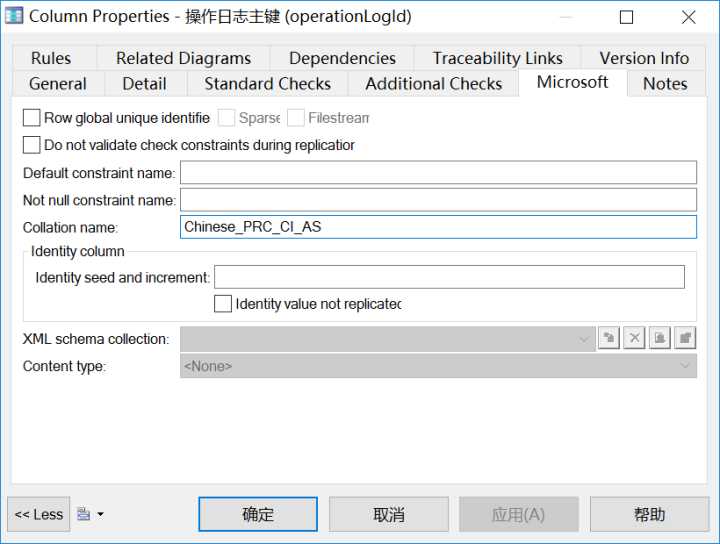
  


其实Character Set和Collation本就是一体的，所以其实都是在这一个地方设置，两种选项对应的SQL语句就是：

ALTER SCHEMA `bsctelmed` DEFAULT CHARACTER SET utf8 ;

ALTER SCHEMA `bsctelmed` DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8\_bin ;

在SQLServer 2008中，只找到了从数据库层及具体字段层面直接配置Collation的方法，但是从表层面中的配置却没有。和MySQL一样，在PD中未能找到全局设置Collation的方法，只找到了具体到某一字段设置的地方。截图如下，最后一张是PD中对某一具体字段进行配置的方法：

如果想查看SQLServer的版本、字符序等相关信息也可以用如下SQL语句：

SELECT

SERVERPROPERTY(N'Edition') AS Edition,

SERVERPROPERTY(N'ServerName') AS ServerName,

SERVERPROPERTY('ProductVersion ') AS ProductVersion,

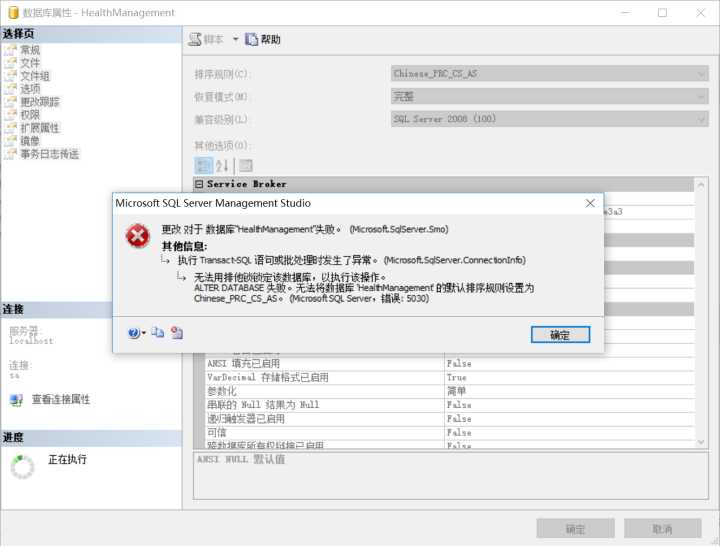
SERVERPROPERTY('ProductLevel') AS ProductLevel,

SERVERPROPERTY('ResourceVersion') AS ResourceVersion,

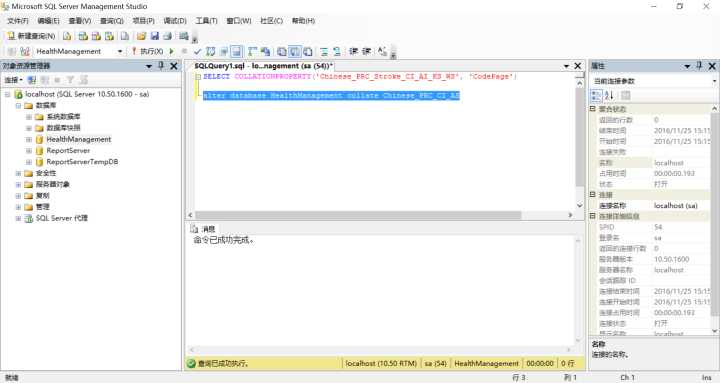
SERVERPROPERTY('ResourceLastUpdateDateTime') AS ResourceLastUpdateDateTime,

SERVERPROPERTY('Collation') AS Collation;

直接在SQL Server Management Studio 图形化界面中更改SQLServer的字符集可能会出现问题：

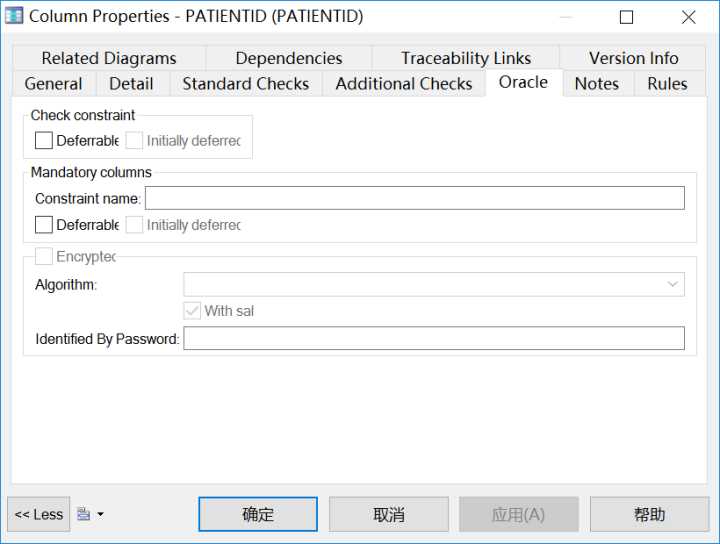


这时可通过如下更改语句进行更改：



ALTER DATABASE HealthManagement COLLATE Chinese\_PRC\_CI\_AS

在Oracle中貌似没有Collation的概念，又或者是换了另外一个概念来进行类似的设置？在PD中具体到表字段的Oracle选项卡中也没有字符集相关的配置：



Oracle中默认是严格区分字段内容大小的，如果不想对大小写进行区分可以使用Lower()或Upper()函数来达到目的，也可以使用NLSSORT（）函数，觉得这个函数就和MySQL中的Collation设置所达到的效果相似。如下三个SQL语句所达到的效果是一样的：

select \* from user where user\_name = 'MengXianzhi' COLLATE Latin1\_General\_CI\_AI;

select \* from user where Upper(user\_name) = Upper('MengXianzhi');

select \* from user where NLSSORT(user\_name,'NLS\_SORT = Latin\_CI') = NLSSORT('MengXianzhi','NLS\_SORT = Latin\_CI');

但是不清楚上面第三种写法，如果不是精确查询，而是模糊查询，用like关键字，查询语句如何下，尝试下面的写法，好像不起作用：

select \* from user where NLSSORT(user\_name,'NLS\_SORT = Latin\_CI') like NLSSORT('Meng','NLS\_SORT = Latin\_CI')+"%";

Oracle9i之前，中文是按照二进制编码进行排序的，而在Oracle9i中新增了按照拼音、部首、笔画排序功能，使用方法如下：

按汉字拼音排序：

SELECT \* FROM USER ORDER BY NLSSORT(user\_name ,'NLS\_SORT = SCHINESE\_PINYIN\_M')

按汉字笔划排序：

SELECT \* FROM USER ORDER BY NLSSORT(user\_name ,'NLS\_SORT = SCHINESE\_STROKE\_M')

按汉字部首排序：

SELECT \* FROM USER ORDER BY NLSSORT(user\_name ,'NLS\_SORT = SCHINESE\_RADICAL\_M')

注意，我虽然严格测试过MySQL、SQLServer和Oracle三种不同关系型数据库针对CharacterSet和Collation设置的区别，但对于同一数据库的不同版本间的区别却未深究。各种关系型数据库总是在不停升级，某些升级可能会导致新旧版本间差异巨大 ，而本文档中所述细节又甚多，所以具体到实际情况，某些地方出现不同也很正常。

C.由此引出的乱码问题

Character Set和Collation并不仅仅影响到数据库存储内容的大小写敏感问题，还会影响到数据库操作中常见的乱码问题。这里既然提到了，所以简单讲下。

以往负责的项目较为杂乱，所以对各种常见关系型数据库多有接触，就自己的经验，乱码问题出现最多的是MySQL数据库，尤其是早期版本的MySQL，其次是Oracle，SQLServer当然也有，但相对少。乱码问题可以分为以下几种：

a． 不同类型的关系型数据库间、数据互相导入导出，导致的中文数据乱码。

比如将MySQL中的数据导入到SQLServer，将SQLServer中的数据导入到Oracle。这种情况其实相对少见，因为一般数据操作都是在同一类型的数据库间进行。遇到这种情况时，数据间的导入导出一般都有中间过程，比如先从源数据库中将数据导出成CSV文件，然后再将CSV文件导入到目标库。又或者是，借助目标数据库的管理工具，直接连接源数据库进行导入。也有将源数据库中的数据导出成SQL文件，然后对SQL文件进行一定更改后在目标数据库中执行的。

b． 类型相同、版本不同的关系型数据间的数据导入导出，导致的中文数据乱码。

c． 类型相同、版本相同的关系型数据间的数据导入导出，导致的中文数据乱码。

d． 针对Oracle，客户端版本和服务端版本不同所致。客户端的版本比较新、而服务端比较旧，或者是客户端为32位的而服务端是64位等箸。

e． 主要也是针对Oracle，客户端和客户端所在操作系统不协调、服务端和服务端所在操作系统不协调。比如操作系统为32位，但下载的客户端却是64位的。

f． 针对程序，官方管理工具操作数据库查询没有问题，但是程序访问数据库查询出的中文却是乱码。

中文乱码问题在各种数据库的操作中、在种程序语言各种项目的开发中时常出现，针对以上种种我们建议：

a． 安装及操作数据库时，编码相关的默认设置，除非有把握，否则不要随意更改；

b． 项目开发环境、测试环境、模拟环境、真实环境、线上环境的操作系统及数据库等尽可能统一版本统一配置，选择和操作系统相匹配的数据库版本；

c． 针对Oracle数据库，客户端和服务端尽可能统一版本，尽可能选择和操作系统相匹配的客户端及服务端版本；

d． 在数据库日常操作过程中均使用官方的管理工具，或直接在命令行中操作；SQLServer不用说，MySQL我们建议使用MySQL Workbench，Oracle我们建议使用SQL Developer；

e． 如果现实情况不方便或不允许达到以上要求，或者虽然按以上要求操作配置数据库后依旧出现乱码问题，那就根据实际情况网络搜索寻求相应解决方案。

遇到具体问题时刻记得Google是第一位的，仅这一项就可以帮我们解决99%的问题。我们的分析讨论建议更多的是为了全面了解问题本身，但遇到具体问题如何解决，仍旧要靠自己思考、靠Google的智能搜索。下面的截图来自于以往笔记，和某一乱码问题的交锋：



D.表名字段名等大小写

上面讲字符序的大小写敏感，针对的都是数据库表字段值或者说是字段对容，而对于数据库名、表名、字段名、变量名、执行目录名等（在执行SQL查询时）的大小写敏感呢？

在Linux下MySQL的数据库名、执行目录名、 表名、表的别名、变量名默认是严格区分大小写的，数据库名大小写敏感不可改，执行目录名大小写敏感可参数调配（lower\_case\_file\_system），表名大小写敏感也可参数（lower\_case\_table\_names）调配，但不确定这个参数是否影响表别名及变量名的大小写敏感。列名与列的别名在所有的情况下均是忽略大小写的，也不清楚可否参数调配。

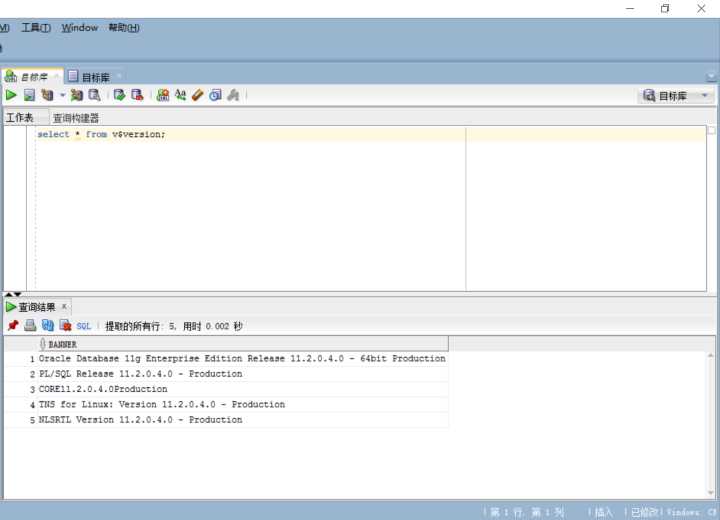
MySQL在Windows下数据库名、执行目录名、表名、表的别名、变量名、列名、列别名等默认都不区分大小写。

用root登录服务器修改 /etc/my.cnf配置文件，在[mysqld]节点下，加入一行： lower\_case\_table\_names=1 可以另其不再区分表名大小写。而在 Windows系统下， lower\_case\_table\_names参数缺省设置即为1，即不区分表名大小写。

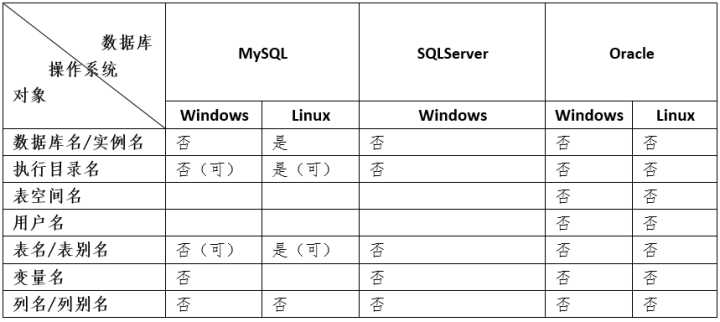
在SQLServer中自己测试的结果是，数据库名、用户名、表名、表别名、列名、列别名默认在执行SQL查询时均不区分大小写。SQLServer版本为2008 R2。



在Oracle中自己测试的结果是，实例名、表空间名、用户名、表名、表别名、列名、列别名默认均不区分大小写。Oracle为Linux版本，11.2.0.4。

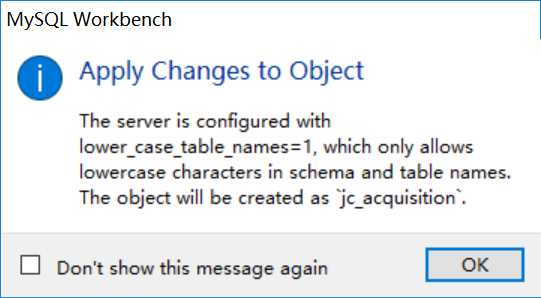


这样整理之后，如下表格：



内容中的是否表示默认情况下是否大小写敏感，括号中的（可）表示可参数调配。空白部分表示不确定，或者没有这一项。

SQLServer和Oracle 、MySQL（Windows系统下）虽然同样默认对表名、字段名等不区分大小写，但不同的是Oracle及MySQL处理的更严谨。通过SQL\*Plus、PL/SQL Developer或SQL Devloper在Oracle中建表，默认会自动将表名转换成大写后再写入数据库。 在Windows系统中，默认情况下，建表时MySQL会强制要求所有表名和列名均为小写。SQLServer虽然在执行SQL查询时不区分表名、列名大小写，但在命名及在可视化管理工具中显示时却又区分大小写。也有另外一种可能，目前我测试用的Oracle及MySQL版本比较新，则SQLServer则较旧，最新版的SQLServer或许已经没有这种问题。



前面说通过SQL\*Plus、PL/SQL Developer或SQL Devloper在Oracle中建表，默认会自动将表名转换成大写后再写入数据库。但实际上Oracle是可以支持大小写混排的命名方式的，但前提是要在表名外面加双引号。

仔细查看过，使用PD设计针对Oracle的PDM，如果你的表名全部大写，那PD在生成SQL建表语句时不会在表名外面加双引号，可如果你的表名是大小写混排的，那PD在生成SQL建表语句时会自动在表名外加双引号，保留这种大小写混排的命名方式。其实不光是创建表，在Oracle中创建触发器、序列时也是如此，名字不加引号就不会区分大小写，加上引号就会区分。

不建议在Oracle中使用大小写混排的命名方式，原因有很多：

a． 当你使用Oracle SQL Developer工具查看表时，点选“详细资料”选项卡，可能会报错：执行请求的操作时遇到错误，ORA-00904:"STATUS":invalid identifier。网上搜到ORA-00904错误原因和Oracle建表时表名大小写有关，但不清楚和Oracle版本有没有关系。

b． 如果表列名都区分大小写，那在建立查询时表名和列名都应该带有双引号，会给后面程序的编写带来麻烦。如果使用Hibernate框架，那其生成的查询是不会带有双引号的，会出现无法找到表或视图的错误。

c． 使用PL/SQL Developer工具可视化地进行表的删除等操作时，后台采用的是不带双引号的表名，也会出现无法找到表或视图的错误。这时只能采用类似 drop table "tableName" 的语法，在SQL\*Plus或PL/SQL Developer手工删除或修改表。

我们在基本规范中为什么要求MySQL的数据库名、表名、列名等统一为小写，Oracle中的表名、字段名等统一为大写，正是基于以上原因。我们希望藉此规定，将命名大小写规则统一，尽可能的让数据库设计不要在名称大小写这个问题上多出不必要的麻烦。

这里顺便一提，在PD中可以将PDM中的表名或列名统一转换成大写或小写，菜单Tools——Model Options——Naming——Convertion——Table或Column中进行设置。

E.针对大小写合理建议

个人认为Oracle数据库对表名、字段名、字段内容等大小写敏感的默认处理是最合适的，在执行SQL查询时不区分表名、表别名、列名、列别名的大小写，但严格区分字段内容的大小写。也正因此，我们在基本规范中建议在Oracle数据库的设计过程中表、字段等的名称统一使用大写，单词间用\_下划线分隔。

我们在基本规范中建议，MySQL数据库、表、字段等名称统一使用小写，单词间用\_下划线分隔。同时，我们建议在MySQL数据库中将Character Set设置为utf8、将Collation设置为utf8\_bin，并在数据库配置文件中设置lower\_case\_table\_names=1，当然，Windows系统中默认就是此种设置，无需再做更改。

我们建议在SQLServer中将排序规则设置为Chinese\_PRC\_CS\_AS，其默认为Chinese\_PRC\_CI\_AS，因为SQLServer数据库不用考虑部署在不同系统的问题，所以不建议更改除此外的其它编码、字符序相关的默认设置。我们上面也说过SQLServer虽然在执行SQL查询时不区分表名、列名大小写，但在命名及在可视化管理工具中显示时却又区分大小写，为了查看方便所以我们在“基本规范”中要求SQLServer用Pascal的命名方式。

在“名大小写”这个章节，更多的不是制定规范，而是在讲解前面的“数据编码”、“基本规范”等模块中列出的一些规范制定的原因。在这里详细讲解了MySQL、SQLServer、Oracle三种数据库的编码、字符序相关的配置说明以及表名、字段名、字段内容等大小写敏感的控制处理等。

**4、具体规范**

A.关于数据库的命名

对于数据库的命名不做特别要求，简单明了即可，这里主要注意在一个大环境中相似项目的数据库命名，最好有明显区分。

这里顺带一提，互联网公司的数据库一般分为五个环境：

a． 开发环境（Development Environment）。开发可读写，开发人员可以修改表结构，可以随意修改其中的数据；但是需要保证不影响其他开发同事。

b． 测试环境（Test Environment）。开发可读写，部署的测试系统访问此库，代测试人员使用。

c． 模拟环境（Simulation Environment ）。 开发可读写，通过web平台，发起上线请求时，会先在这个环境上进行预执行，这个环境也可供部署上线演练或压力测试使用。

d． 线上从库（Real Environment）。 只读，会实时从线上数据库同步，不允许修改数据，不允许修改表结构。供线上问题查找，数据查询等使用。

e． 线上环境（Online Environment）。开发人员不允许直接在线上环境进行数据库操作，如果需要操作必须找数据库主负责人，并做相应记录。

在这些环境中，一定要做到权限划分明确，读写帐号分离，并且有辨识度，能区分具体业务。例如用户名w\_wap、r\_wap 分别表示对wap数据库进行读、写的帐号。

做企业内部应用系统，要求不是特别严格的话，没有模拟环境和线上从库。而且通常情况下，线上环境的库在客户那边，开发测试的环境在公司这边，两边还不能互通，有时不得不驻场开发直接连接线上环境。但是对于线上环境的直接操作是非常危险的，且容易导致线上环境和开发测试环境表结构的不同步，这个一定注意。客户那边应该用权限严格限制对生产环境访问的人员，开发人员自己这边要时刻做好数据备份工作，并提前准备好数据出现意外更改或丢失情况的应对措施。同时，在现场开发，针对线上环境的更改要实施同步到公司的开发环境中。线上线下的所有更改，都要经过数据库主设计师的审核同意。

我们建议，如果可以控制的话，则在不同的数据库环境中统一表空间名、数据库名等，甚至是数据库访问的账号名、权限也可以统一，这样在部署项目时，配置文件则无需再做过多更改，不同数据库环境间有表结构或数据的移植时也可避免出现不必要的问题。在对这些环境的数据库进行备份时，建议在备份文件名中加上前缀和备份时间，以防混淆，比如备份开发环境的数据库可命名为：DevelopmentEnvironment201703271149。这些都是非常细节的地方，有点吹毛求疵，不做强制要求。

B.数据库功能块概述

在前面“设计规范”——“基本原则”——“高级功能”中提到过，现有的开发模式，数据库只用来做数据存储。一直坚持业务相关的部分都由程序处理，不到不得以的情况下不要在数据库中建存储过程、触发器、函数、序列甚至是视图等，尽管如此，这里还是会简单介绍下这些高级功能使用时的命名方式。下面的表格列出了数据库所涵盖常见功能元素的英文名称及缩写：



有建议，除表和表字段外，其它功能块在命名时均要加英文缩写前缀。但就个人意见，除视图外，其它部分加不加前缀不太重要，视图加前缀是为了在执行查询时和表区分开，而存储过程、函数、约束等，我们一眼即可看出它是什么，更何况在可视化管理工具中，这些功能块本来就是各自独立展示的。所以本规范中不强制要求在这些功能上加前缀，但如果要统一加的话，建议使用上图表格中的英文缩写。

C.关于数据表的命名

关于表的命名，TB这种前缀是毫无意义的，本来就是一个表，为什么还要说明？这也是我上面不建议在其它功能块中加前缀的原因。如果表格数量较少，后期项目扩展升级的可能性不大，也没有必要加其它前缀。但有时规模相对庞大、业务逻辑相对复杂的项目，表格数量多到一定程度，在可视化管理工具中查阅浏览不太方便，这时，根据业务或功能对表格进行分类，加前缀也就有必要了。个人感觉是50张表内的数据库，加前缀意义不大，超过100张，则很有必要加前缀。而且我们要求，为了不给后期代码生成造成非必要麻烦，如果要给表加前缀，则所有表均要有前缀，不要出现有些表有、有些没有的情况。

表前缀主要是为了区分不同功能的表，而非解释表的功能，表的功能由表名来解释。前面要求表名的长度要控制在30个字符以内，在此前提下，为了尽可能不影响表的命名，表前缀应该越短越好。我们建议表前缀控制在两个以内。具体表前缀添加规则建议如下，括号内的单个大写字母表示要添加的前缀。这里以Oracle数据库为例，具体表名、前缀的大小写根据实际数据库参照“命名规范”——“名大小写”章节的说明：

a． 系统表（S\_）：System，系统配置相关的基本信息表。系统用户表（S\_USER）、系统角色表（S\_ROLE）、系统菜单（S\_LINK\_MENU）、操作日志（S\_OPERATION\_LOG）、登录日志（S\_LOGIN\_LOG）、系统字典（S\_DICTIONARY）、系统字典类型（S\_DICTIONARY\_TYPE）等。

b． 字典表（D\_）：Dictionary，非系统字典外的字典表。在“设计规范”——“相关注释”——“字典字段”中提到过字典表的定义，除了数据库中的通用字典表，还有一些常见表，比如地区表（D\_REGION）、ICD编码（D\_ICD）等，也是一种字典表，这里的D\_前缀即加在这类字典表名前面。

c． 中间表（R\_）：Relationship，多对多关系中间表。具体命名方式建议为：R\_主表名\_从表名，在多对多关系中其实不分主从表，这里我们规定核心表为主表，另外一个为从表。比如用户角色关系中，用户表（S\_USER）为主、角色（S\_ROLE）表为从，那中间表就命名为R\_USER\_ROLE。当中间表名超长时，则根据实际情况缩写主从表名，建议优先缩写从表表名。

d． 业务表（B\_）：Business，核心业务涉及的基本信息表。这里的业务是非系统配置业务相关的，比如登录、注册、权限这些业务涉及的表都是和系统配置相关的，前缀应该是S\_，而非B\_。比如在线商城的项目中订单业务涉及的表即是核心业务表，会诊系统中会诊单业务涉及的表即是核心业务表，如果项目庞大，涉及业务较多，可以在B后面继续加单字母区分不同的业务，BA\_、BB\_、BC\_……，没必要非得和某个英文对应，只是个代号，和项目组的人员说明即可。

表名前缀的说明如上，已经足够明确，除此外还应该避免无谓的表格后缀。比如存储客户信息的表直接命名为Guest而非GuestInfo，存储航班信息的表直接命名为Flight而非FlightList。还有命名表时，一律使用单数形式。例如，使用 Employee，而不是 Employees，总之，表的命名应该简单明了。

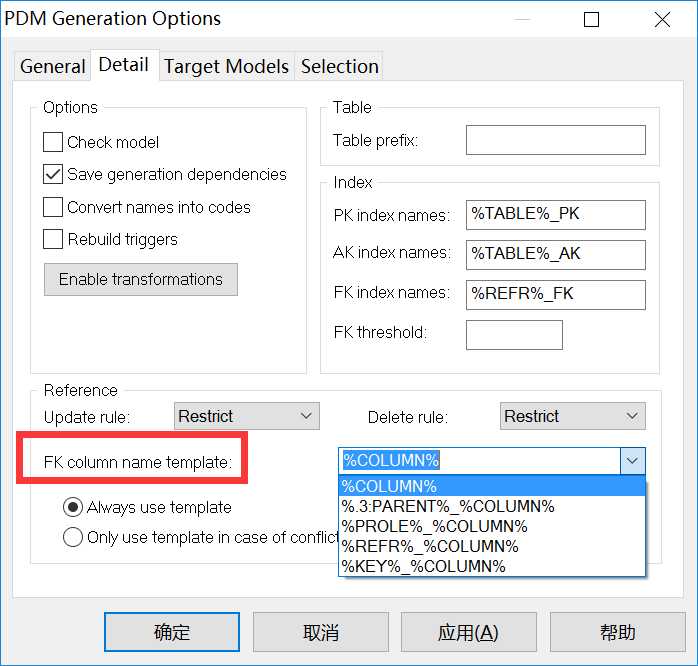
D.关于表字段的命名

a． 所有表中的主键统一命名为id，主键统一使用UUID，类型统一为char(32)。 不建议使用复合主键，即便是在多对多关系的中间表中，个人还是建议用单独的字段做主键，复合字段加惟一约束。

b． 所有的表字段中，除外键，其它字段名都无需刻意加前后缀，也不要在字段名前出现表名。这里的外键是广义上的外键，不仅包括从表引用主表主键的外键字段，还包括存放主表相应关键信息的扩展字段。

比如病人表（Patient），主键就是id而不是pateint\_id，名称就是name而不是patient\_name。但对于外键，比如其它表引用Patient表的主键那就是patient\_id，对应Patient表的name字段那就是patient\_name。如果一个表中有多个外键（字段）同时引用（对应）一张表的同一个字段，那再用其它标识，比如在“设计规范”——“基本原则”——“主键外键”中提到的会诊单申请表中会诊发起医院（sender\_hopital\_id）和会诊接收医院（receiver\_hospital\_id）。

在前面的“设计规范”——“基本原则”——“主键外键”和“设计规范”——“约束控制”中有提到主键字段和外键字段的命名 ，这里再次做以上说明。另，PD中在由CDM转换成PDM时，会自动根据引用关系在从表中添加外键字段，可以自定义外键名称的命名规则：



c． 在前面的“设计规范”——“基本原则”——“连接查询”和“设计规范”——“相关注释”——“字典字段”有关于字典字段的详细介绍，这里再次说明其命名方式：对于字典字段，编码字段后面跟Code后缀，文本字段跟Text后缀，比如gender\_code、gender\_text。

d． 本规范中要求所有表示日期时间的字段，都要有后缀，如果只精确到天则以Date为后缀，如果要精确到时分秒那就用Time作后缀。在“设计规范”——“字段设置”——“通用字段处理”中有关于日期时间类型设置的说明，要求日期时间类型的字段，尽可能精确到时分秒，即便是像生日（birth\_date）这种字段，一般只存储到年月日，但在选择字段类型时建议还是为datetime而非date。所以这里的后缀并不是和具体字段类型对应，而是根据实际业务情况，这个字段存储的数据多是精确到年月日还是时分秒，则后缀相应的为Date或Time。

网上有建议说，日期时间不要用Time做后缀，因为Time还有一个很常用的意思，就是次数。比如登录日志表中有用户最后一次登录时间字段login\_time，不去看表的内容，很容易将login\_time理解成登录的次数。这里我们不予考虑，只要内部统一规范，这就不会是个问题。

e． 本规范中建议是否注销、是否成功等类似的布尔型字段，名称前统一加is前缀，比如是否成功（is\_success）、是否注销（is\_active）、是否显示（is\_display）等。

f． 关于一些通用字段的命名方式建议如下，仅作参考：

E.关于约束控制命名

在“设计规范”——“约束控制”中介绍过五种约束类型：唯一性和主键约束、外键约束、检查约束、空值约束、默认值约束， 本规范中仅对外键约束的命名做要求，因外键约束标明着表与表之间的关系。我们建议外键约束以fk做前缀，后跟从表名称和主表名称：fk\_从表名\_主表名。这种定义方式，约束名称很容易超长，比如在Oracle中，约束名称的长度限制和表名一样，不能超过30个字符。如果超长，我们建议从后向前自动截取多出部分。前面提到过，CDM转换成PDM时会自动根据引用关系在从表中添加外键字段，外键名称的命名规则可以自定义。外键约束名称没必要手动添加，在PD的PDM图中选择：Database——Edit current DBMS——General选项卡——右侧树形菜单 Script\Objects\Reference\ConstName ，在里面可以编辑ConstraintName的命名方式，交由PD自动统一处理，比如可设置为：FK\_%.U30:CHILD%\_%.U30:PARENT%。此设置在PD 15中起作用，16版本中的设置没找到。

其它四种约束的命名，本规范中不做要求，窃以为这些约束怎样命名也不太重要，如果需要统一命名规范， 有些也可借助PD工具进行统一设置。



F.其它功能块的命名

前面说过，因为自己所主张的开发模式，以往的项目中很少在数据库中建存储过程、触发器、函数、序列、事件甚至是视图等，这里只根据经验，给出少量建议。

视图的命名和表的命名有很多相似点，但认为视图的名称最好可直接反应出其查询的主表，或者可明确反应出视图功能。存储过程、触发器、函数、索引的名称则直接反应其功能为好，其命名方式类似于在编程语言中给某一方法命名。序列只在Oracle中有，一般用来填充主键和计数。在早期的数据库设计中，喜欢用自增主键，比如要让用户表（USER）的主键ID自增，则创建名为SQ\_USER\_ID的序列和名为TR\_SET\_USER\_ID的触发器。序列名直接反应出自己要计数的表的列，触发器名直接反应出自己的功能，这种命名方式或可借鉴。

不过后期项目的数据库设计，自己不再用自增主键，原因在“设计原则”——“基本规范”——“主键外键”中有描述。如果项目庞大，数据库设计的模式有变动，要大量使用存储过程、触发器、函数、序列等，对于这些部分的命名还是有必要规范化的。

**5、梳理总结**

“命名规范”牵涉的细节太多，在介绍过程中也一直妄求事无巨细，反而导致有些地方比较散乱，这里把关键部分梳理总结如下：

a． 建议在SQLServer中将排序规则设置为Chinese\_PRC\_CS\_AS，在MySQL数据库中将Character Set设置为utf8、将Collation设置为utf8\_bin，并在数据库配置文件中设置lower\_case\_table\_names=1。

b． 数据库、表、字段等所有名称的可用字符范围为：A-Z，a-z， 0-9 和\_下划线，长度要严格控制在30个字符以内。

c． 数据库、表、字段等所有名称均使用英文单词或英文短语或相应缩写，均使用单数名，禁止使用汉语拼音。

d． Oracle表、字段等名称的统一使用大写，单词间用\_下划线分隔；SQLServer数据库、表等名称采用Pascal命名法，字段名称采用Camel命名法；MySQL数据库、表、字段等名称统一使用小写，单词间用\_下划线分隔。

e． 表主键统一命名为id，主键统一使用UUID，类型统一为char(32)。

f． 表（广义）外键建议命名为：主表名\_字段名，类型和主表中字段类型一样。如果一个表中有多个外键（字段）同时引用（对应）一张表的同一个字段，再根据实际情况加前后缀区分。

g． 对于字典字段，编码字段后面跟Code后缀，文本字段跟Text后缀。

h． 表示日期时间的字段，都要有后缀，如果只精确到天则以Date为后缀，如果要精确到时分秒那就用Time作后缀。

i． 建议是否注销、是否成功等类似的布尔型字段，名称前统一加is前缀，比如是否成功（is\_success）、是否注销（is\_active）、是否显示（is\_display）等。

j． 建议外键约束以fk做前缀，后跟从表名称和主表名称：fk\_从表名\_主表名。

**四 参考文献**

**1、引言**

文档一开始提到过，编写制定这个规范的过程中，从网络上翻阅了数百份数据库设计相关的文章、帖子。并未找到一个大而全又符合心意的，不过即便是耗费诸多时间沉淀整理出的当前规范，依旧有诸多不满意的地方，但就目前来讲，这已经是最适合自己的了。不敢说是目前网络中最好的，但自信这是最认真用心的一份关系型数据库设计相关的文档。

把自己翻阅过的部分相关文章，单独列成出，放在本章节。自己在文档中多有引用，这里对原作者一并表示感谢。

**2、参考文献**

*a． 设计规范*

MySQL 数据库设计规范：[https://loftor.com/archives/MySQL-spec.html](https://link.zhihu.com/?target=https%3A//loftor.com/archives/mysql-spec.html)

MySQL库表设计规范：[http://rainforc.iteye.com/blog/2221285](https://link.zhihu.com/?target=http%3A//rainforc.iteye.com/blog/2221285)

数据库设计规范：[http://www.cnblogs.com/chenmh/p/3944116.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/chenmh/p/3944116.html)

*b． 逻辑删除*

逻辑删除：[http://howiefh.github.io/2015/07/25/logical-delete/](https://link.zhihu.com/?target=http%3A//howiefh.github.io/2015/07/25/logical-delete/)

防用户误删除，耗费一周时间把DeleteMark标志都加上来了，所有的删除操作从“物理删除”转为“逻辑删除”： [http://www.cnblogs.com/jirigala/archive/2010/07/19/1780539.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/jirigala/archive/2010/07/19/1780539.html)

数据库设计里的假删除：[http://blog.csdn.net/macavalier/article/details/6535778](https://link.zhihu.com/?target=http%3A//blog.csdn.net/macavalier/article/details/6535778)

在数据库设计中，当删除一条记录的时候，是加一个标记位还是直接删除这一行：[https://segmentfault.com/q/1010000003938997/a-1020000003975492](https://link.zhihu.com/?target=https%3A//segmentfault.com/q/1010000003938997/a-1020000003975492)

*c． 关系范式*

数据库范式（百科）：[http://baike.baidu.com/link?url=B12kv0sb8nDXeIpvQfobPL3KOuNpM01ihyhZ4oR9sW2bjxnpdG5bATtiQI\_5qweMyL0h97UqmvZlOp7K1537P\_](https://link.zhihu.com/?target=http%3A//baike.baidu.com/link%3Furl%3DB12kv0sb8nDXeIpvQfobPL3KOuNpM01ihyhZ4oR9sW2bjxnpdG5bATtiQI_5qweMyL0h97UqmvZlOp7K1537P_)

数据库学习之范式理解：[http://www.cnblogs.com/sunfie/p/4859294.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/sunfie/p/4859294.html)

*d． 字段类型*

Oracle/SQLServer/MySQL 常用数据库的字段类型及大小：[http://suexiukjung.blog.51cto.com/1169266/1147424](https://link.zhihu.com/?target=http%3A//suexiukjung.blog.51cto.com/1169266/1147424)

ORACLE使用EXPDP和IMPDP数据泵进行导出导入的方法：[http://www.cnblogs.com/peterpanzsy/p/3442784.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/peterpanzsy/p/3442784.html)

char、varchar、varchar2区别：[http://blog.csdn.net/honglei\_zh/article/details/7172538](https://link.zhihu.com/?target=http%3A//blog.csdn.net/honglei_zh/article/details/7172538)

CHAR and VARCHAR Data Types in Different Database Engines：[http://www.vertabelo.com/blog/tec­­hnical-articles/comparing-char-and-varchar-data-types-in-different-database-engines](https://link.zhihu.com/?target=http%3A//www.vertabelo.com/blog/technical-articles/comparing-char-and-varchar-data-types-in-different-database-engines)

MySQL探究之NULL与NOT NULL：[http://blog.csdn.net/fwkjdaghappy1/article/details/7703974](https://link.zhihu.com/?target=http%3A//blog.csdn.net/fwkjdaghappy1/article/details/7703974)

MySQL的空值与NULL的区别：[http://blog.csdn.net/eroswang/article/details/8529817](https://link.zhihu.com/?target=http%3A//blog.csdn.net/eroswang/article/details/8529817)

MySQL的decimal、float、double类型的区别：[http://www.studyofnet.com/news/214.html](https://link.zhihu.com/?target=http%3A//www.studyofnet.com/news/214.html)

MySQL的IP处理函数inet\_aton()和inet\_ntoa()：[http://blog.csdn.net/ssyan/article/details/6199165](https://link.zhihu.com/?target=http%3A//blog.csdn.net/ssyan/article/details/6199165)

Oracle中的decimal与number区别：[http://blog.csdn.net/freshlover/article/details/8809529](https://link.zhihu.com/?target=http%3A//blog.csdn.net/freshlover/article/details/8809529)

*e． 关系约束*

数据库的约束简介：[http://www.cnblogs.com/netsql/archive/2010/05/06/1729162.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/netsql/archive/2010/05/06/1729162.html)

*f． 索引分类*

Oracle索引的分类：[http://blog.163.com/sun\_xiao\_min/blog/static/4859755220118208477618/](https://link.zhihu.com/?target=http%3A//blog.163.com/sun_xiao_min/blog/static/4859755220118208477618/)

Oracle索引分析与比较：[http://tech.it168.com/db/o/2006-08-18/200608181553820.shtml](https://link.zhihu.com/?target=http%3A//tech.it168.com/db/o/2006-08-18/200608181553820.shtml)

*g． 字典设计*

应用开发中数据字典项设计实现方案：[http://junjunbest.iteye.com/blog/395421](https://link.zhihu.com/?target=http%3A//junjunbest.iteye.com/blog/395421)

*h． 历史版本*

数据库模型设计——历史与版本设计：[http://www.cnblogs.com/studyzy/archive/2013/09/09/3310266.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/studyzy/archive/2013/09/09/3310266.html)

*i． PD使用*

使用POWERDESIGNER设计数据库的20条技巧：[http://blog.csdn.net/jiangfeng861016/article/details/7363457](https://link.zhihu.com/?target=http%3A//blog.csdn.net/jiangfeng861016/article/details/7363457)

Quick Reference PowerDesigner 16.1：[http://infocenter.sybase.com/help/topic/com.sybase.infocenter.dc38094.1610/doc/pdf/quick\_reference.pdf](https://link.zhihu.com/?target=http%3A//infocenter.sybase.com/help/topic/com.sybase.infocenter.dc38094.1610/doc/pdf/quick_reference.pdf)

Customizing and Extending PowerDesigner 16.5 SP02：[http://infocenter.sybase.com/help/topic/com.sybase.infocenter.dc38628.1652/doc/pdf/customizing\_powerdesigner.pdf](https://link.zhihu.com/?target=http%3A//infocenter.sybase.com/help/topic/com.sybase.infocenter.dc38628.1652/doc/pdf/customizing_powerdesigner.pdf)

用PD进行版本控制：[http://www.doc88.com/p-806805418378.html](https://link.zhihu.com/?target=http%3A//www.doc88.com/p-806805418378.html)

*j． 命名规范*

MySQL数据库命名规范及约定：[http://www.jb51.net/article/47617.htm](https://link.zhihu.com/?target=http%3A//www.jb51.net/article/47617.htm)

MySQL的Collation：[http://www.cnblogs.com/yjf512/p/4233601.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/yjf512/p/4233601.html)

MySQL中 Character Set 与Collation 的点滴理解：[http://zhongwei-leg.iteye.com/blog/899227](https://link.zhihu.com/?target=http%3A//zhongwei-leg.iteye.com/blog/899227)

MySQL大小写小结：[http://iMySQLdba.blog.51cto.com/1222376/1304539](https://link.zhihu.com/?target=http%3A//imysqldba.blog.51cto.com/1222376/1304539)

MySQL表名不区分大小写的设置方法：[http://www.jb51.net/article/49249.htm](https://link.zhihu.com/?target=http%3A//www.jb51.net/article/49249.htm)

MySQL大小写敏感说明：[http://www.cnblogs.com/wzmenjoy/p/4244545.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/wzmenjoy/p/4244545.html)

匈牙利命名法，骆驼命名法(camel)，帕斯卡（Pascal）命名法：[http://www.cnblogs.com/zsb517/p/4036160.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/zsb517/p/4036160.html)

英语单词的缩写规则：[http://www.360doc.com/content/10/0519/17/289607\_28427480.shtml](https://link.zhihu.com/?target=http%3A//www.360doc.com/content/10/0519/17/289607_28427480.shtml)

C++、Java与C#的命名规范总结：[http://www.cnblogs.com/zyobi/archive/2009/05/21/1486448.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/zyobi/archive/2009/05/21/1486448.html)

Java语言编码规范(Java Code Conventions)：[http://huihoo.org/code/java\_code\_conventions.html](https://link.zhihu.com/?target=http%3A//huihoo.org/code/java_code_conventions.html)

.Net Framework开发人员指南——名称准则：[https://msdn.microsoft.com/zh-cn/library/ms229002(VS.80).aspx](https://link.zhihu.com/?target=https%3A//msdn.microsoft.com/zh-cn/library/ms229002%28VS.80%29.aspx)

C#命名规则、开发习惯和风格：[http://www.cnblogs.com/netshuai/archive/2008/06/29/1231934.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/netshuai/archive/2008/06/29/1231934.html)

Oracle 字符集的查看和修改：[http://www.cnblogs.com/rootq/articles/2049324.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/rootq/articles/2049324.html)

修改SQLServer 的编码格式：[http://blog.csdn.net/zml\_900417520/article/details/50481945](https://link.zhihu.com/?target=http%3A//blog.csdn.net/zml_900417520/article/details/50481945)

查看MySQL和SQLServer数据库的默认编码方法：[https://my.oschina.net/lance4t/blog/136252](https://link.zhihu.com/?target=https%3A//my.oschina.net/lance4t/blog/136252)

更改SQLServer实例默认字符集：[http://www.cnblogs.com/fygh/archive/2012/05/15/2501598.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/fygh/archive/2012/05/15/2501598.html)

如何利用SQL Server2012创建用户自定义数据类型：[http://jingyan.baidu.com/article/6525d4b1699459ac7d2e9486.html](https://link.zhihu.com/?target=http%3A//jingyan.baidu.com/article/6525d4b1699459ac7d2e9486.html)

数据库设计：表的设计命名的十个注意点：[http://yupengcc.iteye.com/blog/2042790](https://link.zhihu.com/?target=http%3A//yupengcc.iteye.com/blog/2042790)

*k． 使用规范*

【MySQL】数据库使用的一些规范：[http://www.cnblogs.com/chenpingzhao/p/5059985.html](https://link.zhihu.com/?target=http%3A//www.cnblogs.com/chenpingzhao/p/5059985.html)