（读书笔记）

XP 的价值和局限

v1.1

ITS 高级会员(http://www.iturls.com/)　张恂

zhangxun2001@hotmail.com

2002/7/5

国际著名杂志 IEEE《Software》2001 年最后一期对极限编程（eXtreme Programming，XP）做了深入报道，一共刊出 4 篇文章，分别是：《从 CMM 角度看极限编程》（Mark Paulk，SEI）[PALK01]、《在一家过程密集型的公司中实施极限编程》（James Grenning，Object Mentor）、《恢复、补救与极限编程》（Peter Schuh, ThoughtWorks）以及《在维护环境下运用极限编程》（Charles Poole and Jan Willem Huisman，Iona Technologies）。作为 CMM 标准制定的核心人物来评价 XP，Paulk 的文章确实是一份非常难得的权威性报告，而后三篇文章则从应用的角度介绍了在不同环境下实施 XP 的成功经验。最后，Glass 先生在“Loyal Opposition”评论栏目中还对 XP 发表了一分为二的看法[GLAS01]。

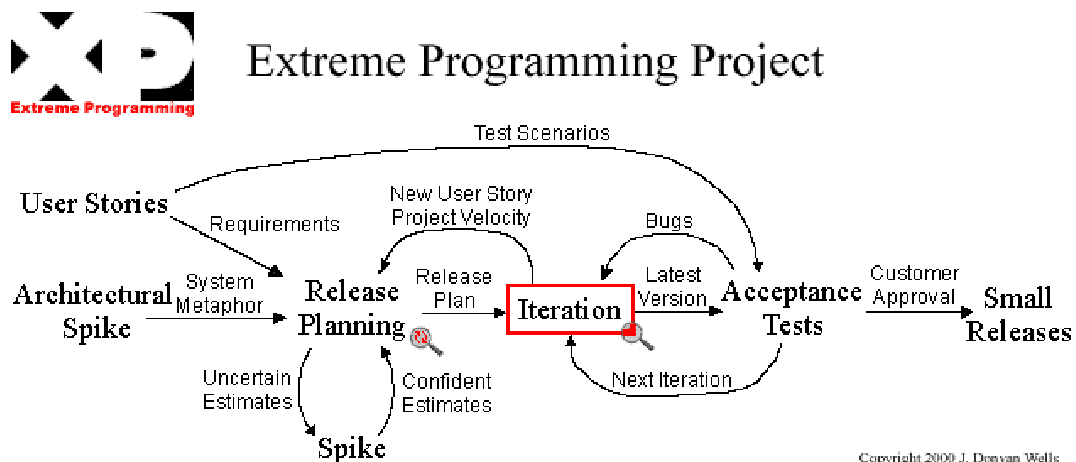
我读了这些精彩的文章，结合自己平日的学习和体会，感受颇多，现整理如下。

ITURLS

什么是 XP？

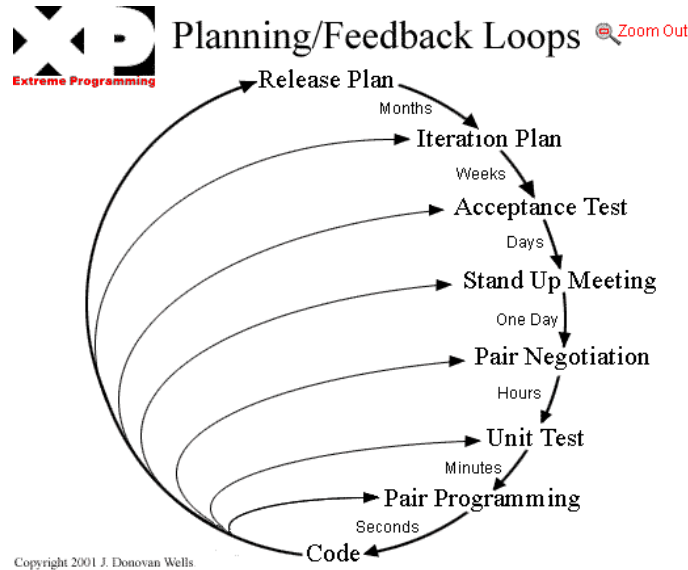
首先回顾一下 XP 的基本内容和特点。Kent Beck 在他的开篇之作《Extreme Programmin Explained – Embrace Change》中提出了“极限编程（XP）”这一创新的过程方法论[BECK99]。XP 是一种高度动态的过程，它通过非常短的迭代周期来应对需求的变化，所以 Beck 把这本书的副标题定为“拥抱变化”。XP 一般适用于需求不确定、变化快，项目历时不超过半年，而人数不超过 10 个、在同一地点工作的中小型团队。

XP 提供了一个全局的、价值驱动的开发过程视图，体现了 4 个价值目标：沟通（communication）、简化（simplicity）、反馈（feedback）和勇气（courage）。XP 的生命周期包括 4 个基本活动：编码（coding）、测试（testing）、聆听（listening）和设计（designing）。一个 XP 项目的状态变迁如下图所示[XPORG]：



XP 的特点是，要求首先开发出最重要的特性，迅速向客户提供所需的功能，随着代码的演进通过重构来满足新的需求，从而使整个项目失败的风险减到最小。XP 的计划/反馈循

环如下图所示[XPORG]。可以看出，从需求定义开始，XP 省略了常规的系统和架构（architecture）的设计步骤，从简短的计划直接进入编码的迭代循环。在 XP 中，编码和设计是同时进行的，而且特别强调测试的重要性。



在一个充满技术、业务和人员风险的环境中，如果对未来投入过多，例如试图通过详细的架构设计、精确的代码结构以及一致完备的文档来满足将来所有可能的情况，结果很可能造成一种精力和时间上的浪费，变化的需求和技术等许多不确定因素会使过去大量的努力付之东流。因此，XP 弱化针对未来需求的设计，非常注重当前的简化。Beck 还建议推迟重要的设计决定，除非某个特性有这样的需要[BECK99]。

既然未来是未知的，需求和环境都在不停地变化，那么只专注于当前已经明确的、优先级最高、最有价值的部分而不必为现在尚且无关甚至可能永远无关的问题进行设计似乎是合理的。然而我认为，XP 有一个非常关键的基础假定，即开发人员只注重眼前需求而依赖将来不停重构所带来的开销和风险要远小于需求变化使事先充分设计失效的代价；如果反复重构的风险要远大于事先进行充分设计的开销，那么实施 XP 可能就是不明智的。这一前提条件决定了 XP 适用的范围和场合。

XP 方法评述

以下结合国内软件开发的实际情况对 XP 的核心内容——12 个实践方法（practices）进行简短地评述。

1）计划博弈（planning game）

XP 要求结合业务和技术情况，快速确定下一次发布的范围。在项目计划的 4 要素（费用、时间、质量和范围）中，由客户选择 3 个，而程序员可以选择剩下的 1 个。通常客户从业务角度确定项目范围、需求优先级和开发进度，开发人员则做出具体的成本和技术估计。XP 强调简短和突发性的计划，有时只用几个小时甚至几分钟就能完成，而且可以随时按需进行多次计划。

我们永远不可能做出绝对正确和完整的计划，因为未来是变化的。最好的解决办法是预见变化、控制风险。如果一次计划做得不够好，那就多做几次。在项目规模小、复杂程度低而不确定因素又多的情况下，XP 的计划博弈确实能够既提高效率又减少风险，但是这不等于不做计划，也不是在做游戏，相反需要良好的技巧。

2）小型发布（small releases）

XP 可以迅速让一个简化的系统投入使用，在非常短的周期（比如 2 个星期）内以递增的方式发布新版本，从而很容易估计每次迭代（iteration）的进度，便于控制工作量和风险，客户的需求和反馈也能够得到及时处理，体现了敏捷的优点。

3）系统隐喻（system metaphor）

XP 通过一个简单的关于整个系统如何运作的隐喻性描述（story）来指导全部开发。隐喻可以看作是一种高层次的系统构想，通常包含了一些可以参照和比较的类和模式，它还给出了后续开发所使用的命名规则。XP 不需要事先进行详细地架构设计。

隐喻在某些情况下确实可以代替正规的架构设计，但它通常只适用于小系统[POL201]。另外，Beck 还提出“不断地细化架构”，这是对的。但是问题出在“不断（ongoing）”上。如果把它理解成鼓励随着项目的发展不断改进架构，是不错；但是，如果像一些人误解的那样，可以推迟架构的分析乃至推迟整个项目的计划和系统设计，那就有问题了。

4）简化设计（simple design）

在任何时候都要求代码的设计尽可能简单，恰好满足当前功能的需要，不多也不少。

5）测试驱动（test-driven）

XP 要求“先写测试，后编码”。失败的测试用例（test case）驱动编码和设计，可以减少不必要的开发量。开发人员必须保证单元测试和集成测试始终运行无误，甚至现场的客户代表也要能够编写功能测试程序。

这是所有软件过程方法都一致推荐的做法。无论怎么强调测试的重要性都不为过。

ITURLS

6）重构（refactoring）

重构是指在不改变系统行为的前提下，重新调整、优化系统的内部结构以减少复杂性、消除冗余、增加灵活性和提高性能。

重构不是 XP 所特有的行为，在任何开发过程中都可能发生。通过重构改进已有代码的设计是对的，但重构很容易被误用。如果像有些人理解的那样，重构意味着在开发的时候进行持续不断地设计（ongoing design），那就有问题了。首先，一个程序员重构完的代码对其他人而言可能并不简单；而陷落于老是重构的陷阱之中会使团队停滞不前，依赖重构甚至会成为轻视设计、把设计推迟到编码阶段乃至发布的最后一刻的借口，这对于大中型项目很可能是场灾难[GLAS01]。

7）结对编程（pair programming）

由两名程序员在同一台电脑上结成对子共同编写解决同一问题的代码。通常一个人写代码，而另一个人负责同时保证代码的正确性和可读性。这可以看作是一种非正式的持续同级评审（peer review）。

两个脑袋是否一定胜过一个脑袋？该做法目前在国内外的争议比较多，它究竟能带来多大好处还有待大量实验的验证。问题的一方面是难以找到合适的人选来实施，很多程序员在工作的时候不希望被打搅，喜欢独立思考[GLAS01]，所以结对编程的两个人的性格和编程技能应该匹配；另一方面，管理层可能不太愿意让两个人来同时完成一件工作，认为这是资源浪费。而国外有些研究则表明，结对编程可以有效地加强协作、促进问题解决、减少软件缺陷、缩短开发周期，在一些项目中获得的收益要大于增加的资源开销[PALK01]。

8）代码全体拥有（collectively ownership）

这意味着任何人可以在任何时候改进系统任何部分的代码。这提高了代码的透明度，增进了团队的合作精神。

虽然此举效率很高，但对于大中型系统，采用该方法若管理不善很可能引起混乱[POL202]。

9）持续集成（continuous integration）

XP 提倡一天之中集成、建立（build）成品系统很多次（每当完成一个任务），而且随着需求改变，要进行不断地回归测试。这不禁令人联想起微软非常成功的每日建立（daily build）和烟雾测试方法。

值得注意的是，XP 的小型发布、持续集成和代码全体拥有都需要良好的软件配置变更管理系统和运作流程来支持。

10）每周 40 小时工作制（40-hour week）

XP 要求尽可能安排程序员每周工作 40 个小时，加班不得连续超过两周，否则反而会影响生产率。疲倦的程序员不可能始终保持峰值效率。

这点体现了 XP 的“以人文本”思想，然而实施起来好像有些困难，过于理想化了，但是如何合理地安排工作量和进度的确值得引起国内软件企业和用户的重视。

11）现场客户（on-site customer）

XP 要求至少有一位实际用户的代表全天候在现场负责确定需求、回答开发团队的问题和编写功能验收测试。

这确实是解决与客户沟通不畅问题的好办法，但是国内许多缺乏技术实力的客户还满足不了这种要求，而且客户往往认为在给出了含糊不清的需求之后就可以撒手不管了，出了问题则要开发者全权负责。可能关键不在于客户是不是一定要到现场。

ITURLS

12）代码规范（coding standards）

XP 强调通过制定严格的代码规范来进行沟通，尽可能减少除代码之外的不必要文档。

XP 对架构描述和文档的弱化对于很多国内的项目可能是个缺点。毕竟代码的可读性不能与图形化的架构模型和规范的文档相比，代码并不是一种有效的适合所有受益人（stakeholders）的沟通媒介[POL101]。

归纳一下，XP 通过以上做法实现了 4 个价值目标：

- 沟通

  让开发人员集体负责所有代码并结对工作，鼓励与客户以及团队内部的不断沟通。

- 简化

  鼓励只开发当前需要的功能，摒弃了过多的文档，坚定地专注于最小化解决方案，做好为新特性改变设计，在系统隐喻和公共代码规范的指导下不断重构的准备。

- 反馈

  通过单元测试和功能测试获得快速反馈。在编码之前先写测试用例，并在设计改变或集成之后重新测试，客户现场代表也能编写功能测试。

- 勇气

  提倡积极面对现实和处理问题的勇气，比如放弃已有代码、改进系统设计。

XP 与 CMM 、RUP 的比较

CMM 告诉组织为了系统化地建立、实施和改进软件开发过程应该做些什么，但没有说明如何去做以及采用哪些具体的技术、策略和方法。CMM 是一套通用的过程实践标准，适用面很广。实施 CMM 要求组织在过程的制度化建设上付出大量努力，通常被认为是重载（heavy-weight）的模型。

XP 是一个针对某种特定环境（需求变化快的小型团队）的具体过程实施模型和方法论。它其实是一种演进式的原型化方法（evolutionary prototyping）[MCNL96]，具有沟通高效、设计简单、反馈迅速等特点，因而是一种轻载（light-weight）、敏捷（agile）的过程方法。

Mark Paulk 在他的文章中把 XP 的实践方法与 CMM 的 KPA（关键过程域）进行了对照。得出的结论是，XP 部分满足或大部分满足了 CMM 2-3 级 KPA 的要求，而基本上没有涉及 CMM 4-5 级的 KPA。这说明 XP 的做法基本符合了 CMM 的目标和 KPA，但还不完备。总体上看，XP 侧重于具体的过程和开发技术，而 CMM 更关注组织和管理上的问题。XP 缺少的一个重要内容是“制度化（institutionalization）”，它不含有被 CMM 认为是使良好的工程和管理实践制度化的关键基础设施和管理要件。[PALK01]

RUP（Rational Unified Process）是一个风险驱动的基于 UML 和构件式架构的迭代递增型开发过程（框架）。RUP 定义了 4 个阶段（起始、细化、构造、移交）和 9 个科目（业务建模、需求、分析和设计、实现、测试、部署、配置和变更管理、项目管理、环境）。这些阶段对应着关键里程碑的划分，而不同科目的工作流和活动在生命周期的迭代中可以并发进行，具体执行的程度则可以调节。RUP 对于角色、流程、工件和活动的要求是灵活的、可配置的，所以它广泛地适用于各种类型和规模的项目。RUP 集中体现了 6 个软件开发的最佳实践方法：迭代式开发、需求管理、构件式架构、基于 UML 的可视化建模、持续校验质量、变更管理。RUP 是一种以架构为中心的开发过程，而这正是大、中型项目成功的关键。

XP 的编码和设计活动融为一体，弱化了架构的概念，这是它与强调架构设计的 RUP 的最大不同。架构的内涵要远大于一些简单的隐喻，它要考虑结构、行为、环境、使用、功能、性能、可靠性、弹性、重用、可理解性、约束和权衡乃至美学等诸多方面的因素。设计架构的目的不是为了完美地表示系统的全部细节，而是为了消除最主要和最关键的架构风险。RUP 细化阶段的主要目的就是构造出一个可运行的架构原型，作为将来添加需求功能的稳固基础。另外，XP 没有包含业务建模、部署等概念，反映了它以编程为中心、节省一切的哲学。

当然两者也有不少共同点。例如，它们的基础都是面向对象方法（取代传统的结构化方法），都重视代码、文档的最小化和设计的简化，采用动态适应变化的演进式迭代周期（取代传统的瀑布型生命周期）、需求和测试驱动并鼓励用户积极参与等等。

由于 RUP 提供了非常丰富的内容，所以常常被误解为一个重载的过程。通过定制 RUP 这个通用的过程框架，去掉项目不必要的工件（artifacts）和中间环节，把 XP 的做法（比如短小的迭代周期、结对编程、测试优先的设计和重构）吸收进来，也可以实现 RUP 过程的敏捷和轻量化[SMTH01]。“Bob 大叔”（Robert Martin）甚至从 RUP 中裁剪出了一个酷似 XP 的最小化 RUP 过程——dx[MART01]。我设想，XP、RUP 乃至其他工程和管理方法可以统一起来使用，姑且成之为统一软件过程（Unified Software Process，USP）。例如，一个大项目团队在起始和细化阶段采用 RUP 方法完成需求分析和架构设计，在构造和移交阶段采用 XP 的做法来实现部分子系统或模块。因篇幅限制，此处不再展开讨论。

“轻载”、“敏捷”是美丽的词汇，无人会拒之于门外。我想 XP、RUP 的目标是一致的——提高团队效率、开发出高质量的软件，而区别就在于各自的侧重点不同，从而导致两者

ITURLS

的适用情况和应用范围有差异。然而，它们是可以互补的，无论是以架构为中心，还是以代码为中心，灵活运用的关键就在于掌握其中的最佳实践方法，实施 RUP、XP 或者融合了两者的过程（比如 USP）都将有助于组织更好地实现 CMM 目标。

XP 在国内应用的问题

那么，国内的软件开发企业应该如何运用 XP，会遇到那些问题呢？

根据 XP 的特点，它尤其适合规模小、进度紧、需求变化大、质量要求严的项目。XP 的一切因变而设，出发点就是希望以最高的效率和质量来解决用户眼前的问题，以最大的灵活性和最小的代价来满足用户未来的需要。XP 在如何平衡短期利益和长期利益之间作出了巧妙的选择。

和任何一种软件方法一样，当然 XP 也不是万灵药。Beck 曾经建议在以下环境中不宜使用 XP[BECK99]：

- 不能接受 XP 文化的组织；
- 中大型（超过 10 个人）的项目和团队；
- 重构会导致大量开销的应用；
- 需要很长的编译或测试周期的系统；
- 不太容易测试的应用；
- 人员异地分布的物理环境。

我想 XP 在国内应用有几种不同程度的实施方式。第一种，全盘接受，严格遵守 XP 的定义执行，估计这种方式不太切合实际；第二种，整个开发过程采用 XP 的生命周期模型，实行 X P 大部分的实践方法，根据实际情况对其中个别做法进行调整或舍弃；第三种，在保持组织原有的开发过程和生命周期模型的情况下，借鉴、采取个别对项目有效的 XP 做法。为了不使管理和沟通效率下降，XP 一般适合于开发小项目的小团队。但根据前面的讨论，XP 其实也适用于开发大项目中的子系统或子模块的小组，这时可以根据该模块在整个架构中的依赖关系来确定该小组的用户和需求。具体在何种环境下采用什么方式来实施 XP 要根据应用类型、项目特点和组织文化而定。

在 XP 的 12 种实践方法中，测试驱动、持续集成、简化设计、代码规范、现场客户、每周 40 小时工作制、小型发布都是我们应该积极提倡和鼓励的，而代码全体拥有、结对编程、重构、隐喻以及计划博弈等做法实施起来要小心，它们并不是在任何情况下都适用的，容易被误用。

目前国内软件开发企业的过程管理水平普遍较低，存在大量轻视或忽视需求、架构、文档、测试以及开发文化和制度建设的现象，许多管理者和开发者不知道在提高开发效率和软件质量方面应该做些什么、如何去做。

举个例子，国内常见的“噩梦”项目一般是这样开始的：草草地做计划、分析用户需求；接着，为了赶工期、赶进度，过分自信地省略必要和清醒的架构分析与系统设计，直接铺开大范围的编码开发，而且基本上不做文档，只在代码中留下一些难懂的注释；然后，没经过充分、认真的测试就急急忙忙推出运行版本；结果，移交阶段软件缺陷层出不穷，麻烦接二连三，再加上业务、市场和技术的不断变化，客户又老是提出新的功能和修改要求，开发人员只好不停地“重构”代码、到处打补丁（“code and fix”），客户却始终不满意；最终，进度一再延误，开发人员疲于奔命，根本没有工夫顾及产品质量，导致客户抱怨不断，项目似乎落入了黑洞……

对比 XP，我们发现，迫于成本和进度的压力，国内的实践其实已经很“轻量化”了——在计划、分析、设计、测试、文档上花的精力和时间很少，但又像实施了半拉子的 XP。

ITUPLS

在各种客观因素和借口的名义下，人们在促进沟通、简化和反馈方面做得很薄弱，勇气却很大，敢于冒由于不规范管理而导致项目失败的风险。指望通过重构来减少投入、改进软件，可是没想到一旦“重构”失败造成的影响和损失反而更大，而依赖于个人英雄和没有测试保证的项目往往是在碰运气。

此外，XP 是一个定义规范的过程方法论。它要求开发团队具备熟练的代码设计技能和严格的测试保障技术，在书中 Beck 甚至想当然地假定如今精妙的技术和程序员的技能已经能够满足实施持续设计变更的要求[GLAS01]；再者，XP 的成功实施离不开高度协同的开发文化和环境。那么，国内的软件开发人员是否都已经真正理解了面向对象和模式，掌握了重构和 OO 测试技术，习惯了“先写测试，后编码”，在技能和心理上做好了准备来获得 XP 的价值呢？

鉴于以上原因，尽管 XP 有敏捷、高效等种种好处，但是现阶段国内的组织误解、误用 XP 的可能性还是很大的。所以，一方面强调提高过程制度化和架构设计的水平，另一方面重视掌握运用敏捷方法的技术和技能，是国内软件开发企业应该面对的双重课题。

结语

在软件学科和许多其它领域中，针对同一个问题，通常至少有两种以上的解决方法。相对于传统的以架构设计为中心的自顶向下过程，XP 方法论的出现则证明了以代码设计为中心的自底向上过程的合理性和有效性，这正是 Kent Beck、Ward Cunningham、Ron Jeffries 等 XP 倡导者在多年理论探索和成功实践的基础上为我们做出的杰出贡献。

成功实施 XP 的关键在于积极有效的管理，对测试驱动编程方式的工具支持，对结对编程的资源保证，用户的积极参与以及对开发人员在项目计划、测试用例编写、重构和结对编程等方面的充分培训。

不过，我对 XP 的名称（“极限编程”）有些不以为然。在软件工程的管理和实践中巧妙地掌握各种平衡往往是成功之道，除了帮助吸引更多的眼球外，“极限”（极端）做法究竟有多少可行性？用“XX 编程”命名一个过程方法也真实地反映了它的局限性。

XP 显然值得我们学习、研究和借鉴。不管当代软件项目符合 XP 实施条件的程度如何，我们都可以从吸收其经验、汲取其价值中获益。实际上，CMM、RUP、XP 都不是全新的发明，它们是对几十年来国外软件开发实践成功经验的总结、继承和发展。但是，如果我们不懂得结合实际、灵活运用反而囫囵吞枣、盲目照搬，就会冒很大的风险。国内软件开发企业尤其应当注重理解、消化这些先进思想方法的外延和内涵，从中学到最佳实践方法，取长补短、持续改进，逐步建立起一套适合自我、行之有效的过程体系。

ITURLS

参考资源

[GLAS01] Robert L. Glass, Extreme Programming: The Good, the Bad, and the Bottom Line, IEEE Software, Vol.18 No.6, Nov/Dec 2001.

[BECK99] Kent Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, Reading, Mass., 1999.

[PALK01] Mark C. Paulk, Extreme Programming from a CMM Perspective, IEEE Software, Vol.18 No.6, Nov/Dec 2001.

[POL101] Gary Pollice, RUP and XP, Part I: Finding Common Ground, the Rational Edge, 2001. (http://www.therationaledge.com/content/mar_01/f_xp_gp.html)

[POL201] Gary Pollice, RUP and XP, Part II: Valuing Differences, the Rational Edge, 2001.

(http://www.therationaledge.com/content/apr_01/f_xp2_gp.html)
[MART01] Robert Martin, http://www.objectmentor.com/publications/RUPvsXP.pdf, a chapter from Object Oriented Analysis and Design with Applications, Third Edition, Addison Wesley, 2001.
[MCNL96] Steve McConnell, Rapid Development, Microsoft Press, 1996 (Chapter 21).
[SMTH01] John Smith, A Comparison of RUP and XP, Rational Software White Paper, 2001 (http://www.rational.com/products/whitepapers/423.jsp)
[XPORG] Don Wells, http://www.extremeprogramming.org/

*感谢您阅读此文，如果您对本文有任何意见或建议，请发信到zhangxun2001@hotmail.com。
**您在“IT 之源”（http://www.iturls.com/）和 umlchina（http://www.umlchina.com/）网站上可以得到本文的最新版本和其他相关资料。

ITURLS