

《0bug - C/C++商用工程之道》

仅以此书，献给我的妻子孙清，你的温柔和善良，使我重新燃起了生活的信心！
此书也献给刚出生的笑笑，你是上天对我的补偿，我期待你纯真的笑容！

肖舸

- 第 1 章 商用工程开发思路
 - 1.1 系统分析初步
 - 1.1.1 需求理解和沟通
 - 1.1.2 “上家”和“下家”
 - 1.1.3 角色“定名”
 - 1.1.4 初步的拓扑图
 - 1.1.5 后续模块级设计
 - 1.1.6 商用设计思维
 - 1.2 商用程序员对开发的理解
 - 1.2.1 资源和成本
 - 1.2.2 盈利导向
 - 1.2.3 客观
 - 1.2.4 平衡
 - 1.2.5 服务
 - 1.3 基本开发思路
 - 1.3.1 边界
 - 1.3.2 “细分”的分析方法
 - 1.3.3 灵活，逆向思维
 - 1.3.4 小内核，大外延，工程库思维
 - 1.3.5 单笔交易失败不算失败
 - 1.4 数据传输各个角色的开发思路
 - 1.4.1 服务器的设计原则
 - 1.4.2 PC 客户端的开发思路
 - 1.4.3 嵌入式设备的开发思路
 - 1.4.4 跨平台软件模块的开发思路

第 1 章 商用工程开发思路

商用工程，一切以需求为导向，因此，系统分析往往是设计工作开始的第一步，同时也是最重要的工作。

就笔者个人的经验，商用工程开发，很多时候不仅仅是技术的工作，更多是人的工作，这要求设计师不仅仅从技术角度考虑项目设计的可实施性，更多地还要根据团队成员的能力，工作态度，学习精神，考虑团队目标的可达性。

这是一个非常细致的工作。

1.1 系统分析初步

有关系统分析的内容，在很多教科书上都有较为详细的描述。系统分析几乎是所有程序开发行为的第一步。但是，商用工程程序员对系统分析应该有一些独特的理解，很多时候，商用系统分析与技术其实没有太多关系，更多的是与沟通和合作有关。

1.1.1 需求理解和沟通

以笔者通常面临的商用数据传输工程来说，这类系统，一般都是指借助网络（局域网或互联网），通过多台计算机的协同工作，共同提供资源，共同分摊 loading，最终为客户实现一个服务需求的商用工程项目。

因此，商用工程的程序员，在接到用户需求的时候，最忌讳的就是马上从编程的角度开始思考，这个功能如何实现，那个模块如何编写。那只会把事情越弄越糟，最后导致不可收拾。笔者一般秉持的习惯是，系统分析期间不涉及细节，先相信所有的细节是能事先的。以后再考虑风险点细节。

提示：程序界很多年以前就在争论，一个程序，自上而下编写（即先搭框架，逐步细化），和自下而上（先解决所有技术难点，做出底层模块，再来拼接），哪个好。在商用工程中，笔者的经验，一定是自上而下，试想，连用什么平台和语言开发都没有确定，如何自下而上？

接到需求，程序员做到的第一件事情，应该是理解需求。大家不要以为笔者在说笑话，在实际工作中，笔者就遇到程序员把需求完全理解反了的例子，还有的程序员干脆挑着看，对于自己熟悉的需求实现得很好，但不熟悉的干脆什么也没做。

因此，那么自己接到的是一个小小的模块，也要认真对待，在理解需求的时候，建议首先仔细看产品相关文档，如需求分析报告，系统设计书之类的文档，然后和自己的上级，可能是组长，可能是部门经理，也可能是项目经理，做一次面对面的直接沟通，讨论一下自己的模块，在未来产品中，究竟处于什么地位，它的优化方向，是空间优先，还是时间优先，有没有特定的算法需求。

提示：现在企业都是团队合作，沟通必不可少，程序员有时候缺乏这方面的主动性，这需要调整。沟通的技巧，要主动陈述自己对模块的理解，言之有物，如“这个模块我的理解是……”，请求上级予以点评，最终确保自己能完全理解，不至于做错事。

另外，沟通时要谦虚，笔者工作十几年，但在上级沟通时，没有听懂，就老老实实说“没听懂”，因为不懂装懂，最后吃亏的是自己。上级一般也不会因此生气，往往是重复再说一遍，确保沟通成功。

1.1.2 “上家”和“下家”

理解需求，还特别需要理解，自己的上家是谁，下家是谁，即自己的模块从谁手里获得数据，自己产生的数据，下一步递给谁。

商用程序模块，原则上“宽进严出”，实现过滤器的效果，即不对上家提出要求，上家给出任何数据，都统统吃下，内部对于自己能处理的数据做严格的校验，不符合的，做出一定的 log 日志之后，予以抛弃，避免错误数据被逐次传递放大，最终形成大问题。

反过来，商用程序模块，在传递给下家数据时，却十分小心，任何一点不对，都可能导致该笔数据被抛弃，道理也是同上。根据经验，秉承这种原则设计的模块，一般都没有较大问题。

在很多时候，我们在项目设计的最后，即将开始 coding 的时候，还玩过“角色扮演”的游戏。即大家一起开会，每个程序员都站到前面来扮演自己所负责的模块，按照每笔业务流程，大家依次口述模块的功能和作用。

比如：从用户输入开始，UI 的程序员说明，我收到用户某笔输入，这代表什么意思，我将其打成什么样的数据结构，并通过什么借口，传递给下面哪个模块处理。对应模块的程序员，立即接上，我收到什么数据，我需要做哪些处理，用途是什么，最终构建什么样的报文，交给哪个模块……

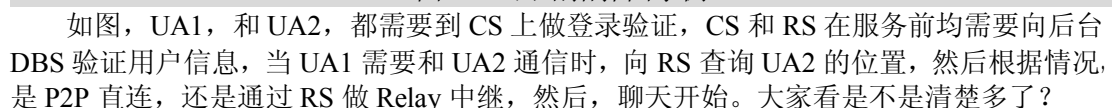
1.1.3 角色“定名”

角色划分,笔者更喜欢称之为“定名”。就是给系统中,互相通信的每一台 Server、Client,定一个合适的名字。名字一旦定出,则每个角色负责的功能,以及其交互的对象,也就呼之欲出了。

我们通常把用户客户端称为 UA 端, 登录验证服务器, 称为 CS, 中心数据库称为 DBS, 用户业务连接握手服务器称为 RS。(S 是 Server 的缩写)

现在我们再看这个 IM 系统，假如一个程序员被分配写 UA 端，他就知道了，原来他一启动，就要求客户输入用户名和密码，然后拿着这些信息，到 CS 上做登录验证，一旦通过，则进入等待状态，如果客户需要和另外一个客户沟通，则通过 RS 查找对方当前的位置，然后试图握手通信，如果成功，则通信开始。最后，客户下线，向 CS 发送下线请求，然后退出。

大家可以看到，一旦定名，对于整个工作逻辑的流程，就有了可描述性。此时，也可以画出一个基本的网络拓扑图（图 1.1）



1.1.5 后续模块级设计

当各个网络角色确定后，下面就是各分项模块的详细分析和设计，这个时候，才正式开始在每个角色内部划分模块。

不过话又说回来，商用数据传输的系统分析，并没有跳出常规的系统分析路子，都是先划分模块，再确定关键数据及其走向，画出基本拓扑图，有必要的话，针对关键业务还要画出时序图，进一步明确动作的协调性，最终完成组织架构的设计。

1.1.6 商用设计思维

系统分析，是一个大话题，业界已经有很多专著来论述如何实现系统分析，本书并不作为重点讨论，因此就不一一赘述了，各位读者有兴趣的话，可以自行参考相关的书籍。

本书在此仅仅强调一点，商用程序员，应该把自己视为系统分析员，再小的模块，也应该秉承系统分析的思路来解决问题。

再举个例子，笔者以前接到一个小任务，写一个 ID 比对的函数，ID 是 1~255 字符长的字符串。这个应用是在一个 RS 服务器上的验证功能。

正常的 C 程序员应该怎么做？strcmp，ok 了，但笔者没有。笔者首先询问了上级和同事，发现这个 ID 是有规律的，同一个组的用户，ID 前 4 位都一样。

那么，笔者就要考虑了，如果使用 strcmp，固然没有问题，但都知道 strcmp，否定优先，即不同的两个 ID 比较，总是比较快的，但 strcmp 又有个缺点，从左往右比较。这意味着，如果是同一个区的用户，前四个字符由于一样，它们的比较时间总是被浪费了的。

于是笔者设计了一个从右向左比较的函数，经过试验，效率远高于 strcmp，使用效果非常好。

大家看出商用程序员的差别没有？

1.2 商用程序员对开发的理解

商用程序员，出于系统分析和设计的需要，对于产品、项目、需求、都有一些相对独特的理解，这些理解和概念，是基于长期从事商用程序开发后，对于市场化、产品化以及抽象化理解。本书中，也大量使用了这种概念，这里提出来，帮助大家对本书的理解。

1.2.1 资源和成本

资源是一个很广泛的概念，在商用程序员眼中，一切都是资源。这不仅仅和技术有关，而且与人有关。

从一个公司讲，市场份额，是资源，客户关系，是资源，公司的流动资金，固定资产，也是资源。

从一个项目讲，投入一个项目的人力，是资源，投入的钱，也是资源。已有的开发软硬件条件，是资源，提供的测试环境，也是资源。

从一个项目团队讲，团队中最顶尖的高手，能实现的极限算法，是资源，团队的整体平均实力，是资源，团队成员的工作时间，也是资源。

从一个程序员讲，自己已经拥有的成熟代码库，是资源，自己的时间，是资源，自己的精力，也是资源。

从一个程序来讲，本模块能使用的 CPU 占用率，是资源，本模块能使用的磁盘空间，是资源，系统设计的目标运行平台，也是资源。

商用程序员，一定是对资源非常敏感的人，因为他们深深知道，资源决定算法，资源决定数据结构，资源决定开发的难易程度，资源决定最终能为公司赚取的利润，以及自己能够赚取的薪水。

举几个例子：

一个项目团队，如果只有一个程序员，那么不考虑大家水平上的误差，这个程序员可以比较随意地决定算法和数据结构，因为都是他一个人写，无需和他人配合。这时候，他可以优先选择自己的成熟代码来完成开发，降低自己的资源消耗。

一个软件系统设计的目标平台很高，使用高端服务器平台，这说明程序的运行资源很多，程序员就可以在开发时，采用一些简单的，便于实现的算法，如使用静态数组，不使用动态内存分配等，以一定的资源消耗，降低开发难度，提升开发效率。

一个商用程序员，做了几年，总有一些相关行业的成熟代码积累，在做分析时，程序员潜意识就会尽量考虑设计向代码靠拢，以便后期实施时降低开发难度和成本。

这些实例说明，不管我们是不是承认，其实，每个人在做事时，都潜意识地会先做资源分析，因为，资源决定了他做事的成本，也决定了他赚钱的速度。

其实，商用数据传输工程，甚至任何一个商用工程的开发，道理是一样的，系统分析，首先就是资源分析，以此来评估项目的研发成本，进而推算一个项目可能的利润，然后决策是否可行。资源分析，其实就是经营中的成本分析。

1.2.2 盈利导向

很多时候，资源决定做事的成本，但成本并不是经营的全部，商业经营中的成本控制，仅仅是手段，最终是为了盈利目标服务的。

这要求商用程序员，不仅仅要做一个技术的高手，还应该深深理解，自己的工作最终极的目标，是为公司盈利。

一个产品，使用再好的技术，但是市场表现不好，不会盈利。一个项目，尽管做得尽善尽美，但超出了工期，客户拒绝买单，也不会盈利。商用工程开发，本质上，是一个经营的过程，其最终目标不是钻研技术，是盈利和赚钱。

举个例子，一个项目，需要处理海量用户的在线数据，作为一个传统的程序员，这么大的量，肯定首先会考虑使用动态的数据管理结构，比如链表之类的算法，由此带来程序复杂度大大增加，bug 率增加，开发的时间成本随之增加，进而，程序员单位时间的产出减少，公司的盈利也减少。

但商用程序员，会首先分析资源，在线数据的信息量不大，每个用户 64 字节估计就够了，一般的系统，用户再多，算 100 万好了，总数据量也就 64M，那，开个 64M 的数组行不行，就在内存里面计算，优化算法也多。最重要的，程序容易，开发快捷，不出错，公司能赚钱。

商用程序员，做事具有明确的盈利目的性，单刀直入，选择算法以最简化为原则，满足需求为原则。并且随时关注资源的消耗，成本的控制。

1.2.3 客观

其实不管是不是做程序，在企业中工作，职业素质的基本要求，就是客观。客观，就是一分为二地看待问题，遇事，既不轻易肯定，也不轻易否定，先分析，再决策。这种态度，不仅仅是技术分析需要，也是基本的做人和做事的态度。

1.2.3.1 做事态度的客观

由于商用项目开发行为，实际上是一次经营行为，有明确的盈利目的，这就要求参与的每一个人，具有较高的职业素质，遇事能冷静、客观滴分析问题，善于一分为二地考虑问题。这是平衡思维。

一个程序员，从学校出来，刚刚走入社会，进入一家公司，一定有很多冲动，希望在软件研发这个领域，能快速证明自己，迅速获得公司和同事的认可，从而获得自己合适的社会地位和相应回报。这没有任何错误。

但是，在商用工程中，由于客户需求千变万化，使用技术无论是深度和广度都远远大于学校中的教学内容，一个新程序员的技能，很难真正满足商用开发的需求，需要大量的钻研和学习。

这就带来两种可能，一种，是新人急于证明自己的能力，遇到难题，一般不会求助与团队，坚持自己钻研，导致开发时间无法控制。另一种，是新人为了表现自己，喜欢在开发中使用一些比较精巧，比较深度的算法来解决问题，但由与算法的复杂度增加，带来了开发时间延长的项目风险。

这两种态度其实都不可取。一个真正的商用程序员，其实是一个善于利用资源的人，他把身边的同事，自己的领导，都视为自己的资源，遇到问题，既不不思进取，轻言放弃，也不会死钻牛角尖，拒绝帮助。

同时，商用程序员在讨论工作时，没有什么面子观念，对于问题冷静分析，客观表述，既不回避责任，也不讳言能力不够，尽量把问题尽早暴露给所有团队成员，帮助团队及时采取策略，规避项目风险，最终实现盈利目标。

举个例子，笔者以前带的团队，有个年轻的伙伴，很喜欢用 C 语言里面的指针，尤其喜欢动态内存分配，经常是指针来、指针去。笔者劝过他，他也不太接受。

后来，在项目开发中，有个数据库的目录服务，交给他写，两个月，没有写出来，总是有 bug，无法完成，笔者审查他的代码，写了 10000 多行，看起来还是很用心。

但后来项目交工期到了，全体项目成员都完工，就等他一个，笔者没有办法，只有替他写了出来，1200 行，什么技巧都没有用，两天时间完工，而且，程序一次过。

最后他被公司请去喝茶，现在，在另外一家公司做程序员了，他跟我说的是，这辈子都不敢再乱玩指针技巧了。

这是典型的乱玩技巧和拒绝帮助带来的恶果，希望各位读者引以为戒。

1.2.3.2 技术的客观

在笔者的程序生涯中，曾经无数次遇到一个问题，就是哪种语言更好。C、C++、JAVA、PHP 等，现在又加进了 Python 等动态语言，甚至还有函数式开发语言。有的是笔者挑来来的讨论，有的则是别人的问题。其实，笔者经过这么多年的分析，认为这个问题，根本不称其为问题，讨论这个问题，根本就是无意义的。

在商用工程中，客户需求千变万化。每一门语言，都有自己擅长的一面，很难讲哪种语言能应对所有需求。但计算机程序设计语言，由于网络，由于跨平台的发展，其界限已经慢慢模糊，接口也越来越容易。

现代工程开发，很多时候，是多种语言的混合体，哪个模块适合用哪种语言，就用哪种语言开发。很多时候我们方案拍板，C 和 C++都输了，Java 或 PHP 胜出，因为开发上层应用，脚本语言的简单、成功率高、开发成本低。

更重要的原因，就是学习成本。有很多主管，喜欢按照自己喜欢的语言来做系统设计，其实是很要不得的。团队是由多个程序员组成的，每个人都有自己习惯的语言，在熟悉的领域，大家成功率都比较高，开发成本也低。如果不顾实际情况，硬性规定一定要用哪种语言，程序员为了完成任务，重新学习，这个学习成本，其实也是项目的成本消耗，很多时候得不偿失。

同样的问题，也体现在算法和数据结构上的选型上，项目团队的主管，系统架构师，很多是技术出身，因此在开发过程中，角色定位不准，经常越俎代庖，替程序员选择算法。布置一个任务，不但要结果，还要求程序员必须用哪种方法做。

每个程序员都有自己熟悉的算法，有自己成熟的代码库，而用其成熟的技术开发，本来就是大大节约成本的一件事。

因此，除非超出程序员能力，完全不能实现需求，一般建议架构师或项目主管不要太关注算法细节，避免效果适得其反。

举个例子：

以前一个项目，服务器的，当时的方案有点问题，项目组最后完成时，测试到并发登录率很低，每秒只能 3~4 人并发登录，而要求的设计指标是 70 左右，全体吐血。

分析原因，当时的 CTO 选择的 qdbm 数据库，根本没有做性能测试，仅仅参考了一份网上的报告，说每秒能支持 10 万次以上的读写，就决定使用。由于他是 CTO，有方案最终裁决权，项目方案就此定稿。

但最后经过我们实测，这份报告，是 qdbm 的枪手写的，这 10 万次以上的读写性能，是 cache 机制全效率运行的效率，换言之，是所有数据都在内存时候的效率。

问题是，我们业务数据差不多 20 亿条，放在内存中，根本就不现实。还有就是，qdbm 的 CPU 占用率太高，那个枪手也很不负责任，根本没有说明，只要达到 1 万次/秒的读写频率，CPU 占用已经到了 90% 以上，其他业务也就不必做了。

而上面这个 CTO 根本没有仔细分析，就决策使用 qdbm。导致项目最终失败。

最后，笔者所在的项目组 12 天，推翻他的方案，使用 MySQL 数据库，更换了数据组织形式，重构了几万行代码，才算按期交活。笔者那 12 天，基本上没怎么睡觉。

这就是一般说的一将无能，累死千军！仅仅这一个失误，公司十几个人做了将近一年的无用功，浪费差不多 200 万人民币。希望各位读者以后引以为戒。

1.2.4 平衡

当我们在学校的时候，出于教学和科研的目的，同时也是因为我们时间丰富，做事情对成本的考虑较少，因此可以把一个算法，一个数据结构，仔细钻研到极致，最终实现最佳的配比性能。

但很遗憾，在商用工程中，这种态度其实是最要不得的。

比如一个客户需求，就要求服务器能承接 100 个客户的并发请求，我们在那里研究算法，做出了 10000 路并发请求的服务器，好不好？肯定好。但是，有必要吗？

客户只要 100 并发，就只会给 100 并发的钱，我们做得更好，多花了很多资源，但没有赚到更多的钱，其实是做无用功。

商用程序员是平衡的高手，他们能在满足客户需求的前提下，尽可能节约成本，另一方面，他们也随时在计算新需求的开发成本，如果感觉某个功能，开发起来成本过高，则会主动提出建议，建议公司的市场人员和客户沟通，取消这个功能，或者换成别的效果差点，但成本很低的功能实现，再或者，请客户多花一点钱，换个更好的运行平台。

这在公司里面做事，其实程序员会面临一些压力，很多时候，市场人员无法理解程序开发的成本计算，可能由于客户难以沟通，简单把这类问题归纳为程序员水平不够，态度不好，并向公司汇报，以此向研发人员施加压力。

笔者在商用工程开发中，这种问题遇到很多。确实没有太多太好的方法，一般说来，笔者会给出详细的分析说明文档，论述自己请求降低需求的理由，另外，笔者长期的工作态度和业绩，也说明了笔者的技术水平，对方的压力没有太大说服力，最终，一般都能得到一个较为平衡的结果。

因此，商用程序员，要关注到自己也是公司的一份子，遇事要多沟通，保持高度的职业化精神，不断关注自己的输出和贡献，创造自己的个人品牌价值。这些，看起来是一些做人的道理，但也是研发工作能顺利进行的保证。

说到平衡，还有一种系统设计平衡，也请各位读者关注，就是在商用工程中，一般很忌讳把某个单项性能做到登峰造极。

原因很简单，一台计算机系统，天生应该是平衡的，CPU 就那么快，内存就那么大，如果其中某个功能，性能做得太突出，太优秀，笔者一般不会认为这是好现象，反而担心它占用了太多的资源，导致其他业务无法正常运行。这不好。

即使计算机系统资源足够，不存在上述矛盾，但是优秀的设计，往往占用程序员更多的研发时间，这也是成本，项目管理者应该加以重视。一般说来，客户要求 100%，我们一般做到整体性能 101%，就够了，千万不要试图去把某个模块的性能做到 150%，这可能会导致其他模块的性能不到 20%。因此，当我们极力研究某个高效算法时，也要综合考虑系统的综合性能，不要太走极端。

这个例子是最近发生的，一个公司的同事，女孩子，过来问我，如果要做一个文件传输模块，是不是需要开线程池等一系列设计，使用多任务方式，并发，获得最高的性能。

笔者仔细询问了她的设计需求，发现这是一个 VoIP 语音系统附带的功能，就是说，这是一个附属功能，系统的主营业务是语音，要确保语音的流畅。根据经验，一旦使用多任务并发，可以很轻松地抢夺网卡的带宽，达到高传输率，但是，也很可能导致语音的品质无法保证。

由于这个系统是以语音为主的，笔者经过思考，建议她简单做个串行队列，慢慢传文件，这个模块，不但不追求效率，而且，每传一个模块，都要可以睡眠 10 毫秒，为关键业务主动让出带宽。

这个设计，获得了项目组的一致同意。

笔者有个经验，一个商用系统设计完，随便拿一个性能参量，比如说就是吞吐率，以时间轴画一条测试曲线，这条曲线，一般应该很平滑，不要有尖锐的波峰和波谷，这是一种比较好的状态。

如果有任何一条曲线，出现了一个波峰，系统总能力又是恒定的话，那么，可能另外的一条或几条性能曲线会出现波谷，这意味着，另外的一些服务，目前无法工作了。

1.2.5 服务

在笔者的职业生涯中，很多次听到一些企业宣称自己是解决方案提供商，这，其实已经是在卖服务了。

我们知道，现在生产力过剩，很多情况下，产品同质化严重。比如简单的一个手机，市场上看到的，就有几百个厂商的几万种产品。消费者如果仅仅是需要打电话的话，可能会挑花眼。

所以，现代企业营销，有句名言：“差异化求生存”。即我们做一个产品，不要求一定比竞争对手做得好，但一定有不一样的地方，这个差异性，是经过细分市场分析后，优选了目标客户群的需求后，产生的差异性。比如，一部为中老年人做的手机和一部为中学生做的手机，显然需求有很大差异。

而在差异化生存中，最大的差异化，莫过于服务品质差异化。有车的朋友，可能有这样的经验，在一个普通的修理铺，和一个 4S 店，虽然都是修车，但获得的服务品质、享受都大相径庭。这就是服务差异化。

因此，笔者认为，商用程序员，要有服务的意识，以差异化方式，逐渐提升个人品牌，最终，从技术上，从职业道路上，走出一条成功之路。

1.2.5.1 做人要有服务的态度

现代商用软件开发，一般都是大工程运作，那种单兵作战的程序员英雄时代，已经过去了。以后的社会，是一个合作的社会。

在公司里面，我们往往是一个团队进行研发，完成项目，在项目过程中，所有的团队成员，都在为同一个目标而奋斗。在笔者看来，商用程序员，作为团队的一份子，是必须要有很强的合作精神和服务意识的。

一个人开发程序的时候，很多读者都有体会，程序的设计规划会显得随心所欲，因为没有什么禁忌和阻碍，怎么选算法都可以。但在项目团队中，一般是没有这么好的条件的。

很多时候，我们的模块，要从上级接收数据，经过自己的业务处理，再交给下级，依次传递，完成工作。这说明，我们写出的程序，不仅仅对最终用户负责，更要对团队成员负责。

商用工程项目，一般都是以成败论英雄，一个软件，最终是以整体的用户体验获得客户的认同，从而销售获得利润，即使是做到了 99%，但一天没有全部完成，一天就不能说项目成功。

因此我们开发时，不仅仅要考虑自己的逻辑正确，功能实现，还必须考虑团队伙伴的工作是否能顺利完成。团队所有成员的工作成果，和团队所有成员的收益是绑定的关系，有一个人玩不成任务，团队全体受损。

因此，团队合作，笔者首先建议大家建立“服务”意识。这些意识包括：

- 1、设计和实施工程项目时，不要眼睛老是看着自己的模块，多想想别人的模块是如何实现的，有没有好的方法让大家工作完成得更好。
- 2、随时考虑“上家”的数据是否有疏漏，帮助上家查漏补缺。
- 3、谨慎地调用“下家”的模块，确保给出正确的数据参数，避免下家崩溃。
- 4、主动热情地和同伴讨论问题，把问题尽量暴露在明处，自己有问题，主动请大家帮助，别人的问题，也主动思考。
- 5、即使是模块级程序员，也试图用系统分析的思维考虑系统整体设计，帮助架构师或主管完善设计，从自己的模块角度提出优化建议，并积极讨论，努力把工作做得更好。
- 6、以较为职业的态度和所有伙伴沟通，讨论问题对事不对人，尽量持客观的态度分析和解决问题，获得团队成员的认同。
- 7、遇到事情勇于承担责任，对工作尽心尽力，一切以完成项目为出发点，尽力使自己和团队完成项目任务。
- 8、高度负责地完成自己的工作，努力做到 0 bug，以较高的产品质量，实现对项目团队整体目标的支撑。
- 9、主动关注其他一些，对项目实施有帮助的情况和风险，及时提出建议，帮助项目团队查漏补缺。
- 10、总之，当我们每个人为团队成员服务好了，我们的产品、项目对客户的服务承诺才有保障。

1.2.5.2 技术上的服务观

在笔者多年的职业生涯中，学到了很多的服务理念，在系统分析时，常常不知不觉也将这个概念带入进来。

做过程序设计的读者应该都清楚，程序设计，现在基本上都是模块化的，程序的运行过程，往往是多个模块互相之间的调用结果。C++ 的类对象，尤其说明这个问题。

但笔者不这么看，笔者认为，程序，其实是一个个的小的服务构成，每个模块，都想外提供一些功能服务，程序的运行，实际上是上次应用不断调用下层服务的一个过程，系统分析，其实也是一个服务细分的过程。

笔者认为这个思路的专门，其实是系统分析做久了，不断使用细分思维的一个结果。根据笔者的经验，在商用工程中，这种服务思维，对解决问题非常有帮助。

我们可以看到，软件系统的每个模块，既是服务者，又是被服务者，每个模块，被多个其他模块服务，也同时为多个其他模块服务，**服务，就是某种资源的拥有者对外提供的一种功能调用，其他模块在需要时，通过对该服务提出申请完成功能，避免自行处理。**

这样，模块处理的数据和逻辑趋于简单化，以简洁的应用接口 `mpi` 或 `api` 来实现互相之间的沟通，彼此既不关心对方的实施细节，也拒绝别人关心，高内聚，低耦合，最终，达到系统高效、快速开发的目的。

如下图所示，所有的模块，没有必然联系，全部靠业务逻辑规定的服务依赖关系，共同完成服务。

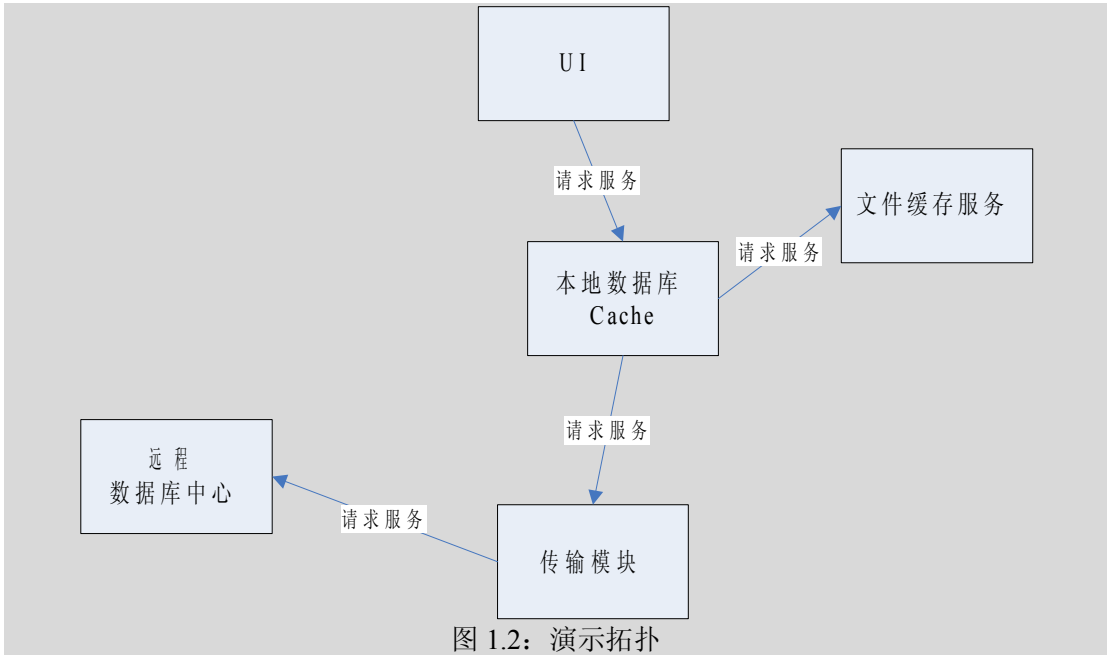


图 1.2: 演示拓扑

同时，从细分服务的观点看，程序上下级逻辑层次逐渐清晰，下层相对上层称为功能层，具体实现一个计算服务，上层相对下层称为业务层，通过组织服务实现自身逻辑，各层之间逻辑清晰，分层明确，形成服务栈的模型，最终实现对客户的一体化服务。

如下图所示，各个模块形成明确的服务支撑栈，每一级，都是上级的功能层和服务支撑层，也是下级的业务层，共同实现最终的客户服务。

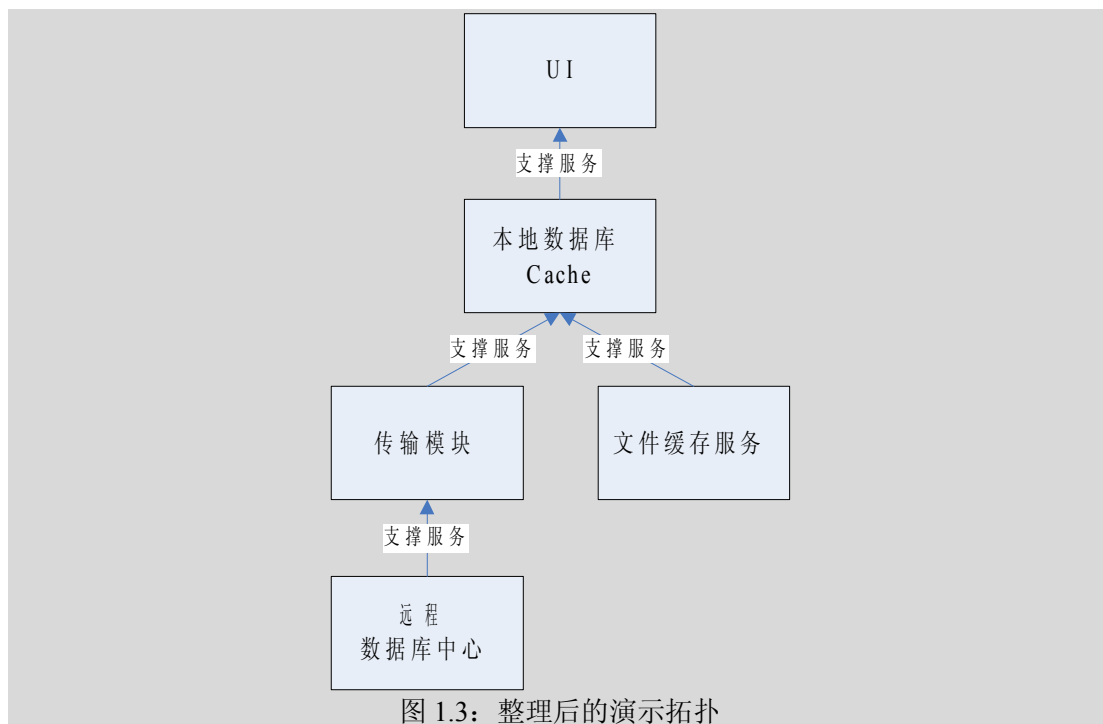


图 1.3: 整理后的演示拓扑

简单说来，在技术上，服务思维，其实也是细分思维，在后文中，有一个具体的实例来讲解如何通过细分服务，来实现业务的优化。

1.2.5.3 Move Loading 问题

商用工程一般是团队开发，因此从服务问题，带出了一个相对很常见，但又很敏感的话题，就是 Move Loading。

前面讲过，Loading，就是服务器的带宽压力，后来又被引申为计算压力，是对资源的消耗。

在团队开发中，往往有这样一些情况，一些复杂的算法，我们俗称“脏活”，由于设计和实现起来，非常麻烦，很多时候，考虑到具体设计人员的工作能力，工作态度等，团队的架构师或者项目经理，可能会刻意把这部分工作，转移到服务支撑链上另外一个程序员处，由别的程序员来开发。

有时候，架构师自己参与开发实施时，也可能通过这种方法转移自己的压力。当然反过来也有，有些架构师，可能考虑到某个项目成员的能力局限，主动帮助该成员完成部分工作，就直接写到自己负责的模块中。

这类行为，笔者一般统称 Move Loading，就是转移压力。请大家注意，这种做法非常不好。

首先，从工作任务安排的公平性上说，接手的程序员多做了很多工作，但从系统的要求看，这部分并不是他的工作，他相当于做了工作，但没有获得应有的承认，也就没有适当的回报，这会造成团队内部很多不满，严重的，甚至会使团队分崩离析。

其次，从技术上说，Move Loading 不但不减轻负担，反而增加 Loading，我们知道，当一个“脏活”在 A 模块，被刻意分配到 B 模块完成的时候，程序必须把该工作相关的数据一起发送到 B 模块，逻辑上多了数据传送这一块，设计上，多了一些信令处理，运行时，多占用了带宽和计算资源，但这个脏活，到了 B 这个模块，该做的事情一点没少。从系统

整体 Loading 上计算，相当于团队、B 模块的实施者、公司、客户都在买单，仅仅是为了让 A 少做事。这显然是错误的。

因此，在笔者主持的团队开发中，这类行为一般是严厉禁止的。如果一个程序员，确实没有能力完成分配给他的任务算法，笔者可能会帮他做，但一定是放在他的模块中，同时，他本人工作量的计算，也相应减少，维持一种必要的公平性。

一个简单的界定 Move Loading 的原则就是“**资源最近原则**”，即任何工作都应该在其需求的资源、或基础数据的最近处处理，因为此处的整体传输成本最低。

比如图 1.3 的例子，文件缓存存取效率的优化算法，一定是放在文件缓存服务模块中完成，不可能由本地数据库 Cache 代为完成。虽然看起来，它们俩挨得很近。当然，数据库 Cache 可能会考虑到文件存储的便利性，以特定的优化格式存储数据，方便文件缓存提升效率，这个不属于 Move Loading，而是系统整体的性能优化。

笔者经历的 Move Loading 最明显的例子是一个服务器集群的开发，开发中，PHP 的程序员需要通过 socket 访问我们的后台数据库服务，那个程序员一直是做脚本语言，对 socket 开发不熟悉，在项目方案讨论会中，就提出数据库服务要像 MySQL 那样，也提供出 apache 的 so 接口，供 PHP 调用，PHP 只管把信令送进去，传输由 C 来完成。

当时大家也没在意，就同意了，项目也顺利完成，但后期测试，感觉到前端的 apache 服务向后台数据库集群提交数据时，显得有点慢。并发量还是很大，但就是每笔交易偏慢，大约 300 个毫秒，有点长了。

笔者百思不得其解，直到项目结束之后，大家聚餐，笔者才想明白这个问题，C 写的 so，里面建立的 socket 连接，由于是被 PHP 单次调用，函数退出，socket 必须关闭。

因此，每笔交易，都需要经过“建立链路 - 传送数据 - 拆除链路”这个过程，socket 没有重用性，大量的时间都浪费在这个连接和拆除 socket 上了。同时，极大地浪费的系统计算资源。显然是个失败的设计。

这根本就是 Move Loading 的行为，因为 PHP 本来就有 socket 的函数支持，如果 PHP 自行完成，由于其内部有共享变量区，可以实现 socket 重用，那么大多数访问，前述的 300 毫秒的交易时长，可以缩减为几个毫秒。

不过可惜，此时木已成舟，笔者也只有留个遗憾了。

1.3 基本开发思路

在商用数据传输工程中，需要的基本开发思路，也就是基本的开发原则，系统分析时的关注要点。这些要点，不仅仅是资源上的关注，需求上的关注，也是思路上的方向。

这里，给大家提示一些商用工程常规的思考方向。

1.3.1 边界

任何系统，资源不是无穷无尽的，因此，任何不涉及边界的系统设计，都是失败的设计。笔者工作这么多年，每次接到需求的第一要务，就是落实边界。

什么是边界？一个系统最多有多少台设备？多少个用户？平时的工作流程，并发访问有多少？每台服务器、每台客户端，最多可以有多大内存，多快的 CPU，硬盘多大？这些是大边界，必须第一时间和客户确定清楚。

原因很简单，**边界决定数据结构，边界决定算法。**

比如，一个只有 255 用户的小型局域网 IM，我们在管理用户时，一个磁盘文件就可以了，启动服务器时，可以直接把这个文件一次读入内存中的一个静态数组，然后使用。

但一个 100 万用户的大型运营系统，可不敢这么干，内存中显然放不下这么多信息，肯定要用数据库来保存用户数据，一旦涉及数据库，就要根据具体业务，设计各种表、查询等数据库相关细节，必要时，查询的信息甚至要做 Cache 系统，以保证效率，等等。。。

二者的开发成本天差地别。

举个例子，以前笔者接到军方一个项目，对方说的很笼统，要一个系统，他们内部可以互相传文件，嗯，还要考虑保密，不能乱传，传输的每一份文档都要有监控。并且还要直接收发短消息，因为军方网络内部不允许使用 QQ，嗯，还要一个网站，有些文档可以放上去，网站上常规的那些新闻什么的都要。文档上去，还要有权限，不是每个人都可以看的。对了，还要一个聊天室，大家可以打字开会。

很乱是不？刚开始笔者都想拒绝了，这个软件的规模，大概需要一个 Exchange + IIS + QQ + 一套网站 + 一套论坛。。。笔者不是神仙，做不出来的。

可是不对，对方给的开发费用只有几万元，和系统规模根本不相称。

于是笔者就专门找到对方的领导，仔细询问了边界，原来对方单位，统共才 100 多人，他们的文档也不多，最多每天 1 篇，每年 300 篇，他们单位总共才 50 多台设备。这个系统主要是应对上级的信息化建设检查，因此网站上功能要全，但对性能要求不大。

呵呵，知道了边界，就好做了，VC 开发一个 IM 软件，最多 256 用户，内部设置一个公告板和聊天室，这个软件包括文件传输功能，传输文件不使用 P2P 模式，全部到 Server 上做 Relay，Server 同时拷贝一份备份，供事后泄密追溯。

找朋友拿个企业网站修改一下，企业改为单位，包括新闻、论坛、公告等简单几个项目。文件分发，在 Server 上开辟 ftp 服务，IM 客户端可以上传文件，文件自动显示在网站中，然后大家可以点击下载。做个简单数据库，记录上传文件名字和上传者，供事后追溯。就这么简单，大家可以看到，边界一旦界定，很多时候，方案马上就出来了。

1.3.2 “细分”的分析方法

在笔者这么多年的开发经验中，深深感觉系统分析中最重要的就是细分，这个概念本来是从笔者学习营销时的“细分市场”得来的，但发现在系统分析中很有用。

我们接到一个客户需求，通常都是笼统的，概括的，模糊的。客户不是专业人士，很难把自己的需求清晰、有条理地说清楚，这就要求程序员有一个细分的思路，要能把客户需求不断拆分，一直拆分到每个小功能，都能落实到程序可实现的地步，而这个过程，其实就是系统分析的过程。

简单的细分，大家可能都会，上面关于边界的例子，其实也是细分的例子，但是，笔者这里强调一个概念，“一切皆可细分”，这才是最重要的。

还是用例子说话：

所有的公网服务的 IM 软件，如 QQ，一般都有用户管理系统，即每个用户需要凭用户名和密码登陆。

我们都知道，IM 软件有好友的，一个用户总有个几十个好友，这就有一笔业务需要处理，每次用户上线，需要查询这几十个好友的在线状态，然后显示到界面上。而且这还没完，好友可能会动态上下线的，因此，每分钟需要刷新一下好友状态。

就这一个看似不起眼的业务，让所有做 IM 软件的程序员痛苦不堪。原因很简单，业务量太大了。

一个同时在线 100 万用户的系统，每个人上线后，每分钟中需要提出一个查询请求，查询几十个人的在线状态，一分钟 60 秒，100 万除以 60 秒，每秒钟差不多接近 17000 笔交易，不要说数据库，传统的 apache 连接能力根本无法承受。

笔者当时几乎什么技术都想完了，Cache，MySQL 事务，等等，但是效果就是不好。

直到最后，笔者突然发现一个问题，我们大家的思路，都局限到中心数据库的星型模型，另外，大家都有个思维误区，认为用户验证管理的 CS 服务器，理所当然应该负担好友查询。

一切皆可细分！

笔者就思考，为什么一定要把这个业务放到中心数据库完成？用户在线查询和用户验证，明明是两笔业务嘛。

思路一旦确定，解决方案就很好解决了，在中心数据库集群开发一个服务，就叫做用户状态查询（GetUserStatus），每个客户端上线，中心数据库在验证通过后，向这个服务发送一个 User 上线的通知，下线也发一个下线通知，然后，客户端所有查询好友状态的请求，从中心数据库转移到这个服务完成，自己不再响应。

一个用户的在线状态，最多几十个字节，算 64Bytes 吧，那么，100 万用户，也才 64M，完全可以放在内存里面，因为这是用户在线的动态信息，也没有磁盘保存的必要。

一旦数据放到内存里面，优化方案就太多了，哈希 Map，平衡二叉树，等等。其效率远高于数据库的磁盘 IO 查询。

客户端根本看不见，一切都在服务器集群内部透明完成。

然后试验，效率比以前提升了 1000 倍，而中心数据库服务器的 loading 马上下降了 1000 倍。完全没有必要再花几十万美金添置高性能数据库服务器集群了。

大家可以看到，简单一个细分思维，把原来大家认为理所当然应该在一起的业务，拆分开，就是大量的成本节约，大家是不是可以学到点什么？

1.3.3 灵活，逆向思维

这其实可以算作细分思维的延伸。

上文的例子，大家可以看到，其实这也是逆向思维的一种体现，细分和逆向思维，本来就密不可分。

笔者这里提出来，主要是提醒大家，不要固执己见，不要墨守成规，很多创新，都是从对大家公认的事中，找出不合理的地方，加以改进而实现的。

笔者工作多年，经常给团队成员讲的一句话：“**在数据传输领域，你亲眼看见的，都不是真的**”。这个话怎么理解呢？还是举个例子。

网络上两个角色，建立起了一个 socket 链路，工作一切正常。那是不是万事 OK？

公网传输，会有多级网关，我们和另外一个计算机进行传输的时候，中间不知道经过了多少路由器或者网关，如下图（图 1.4）

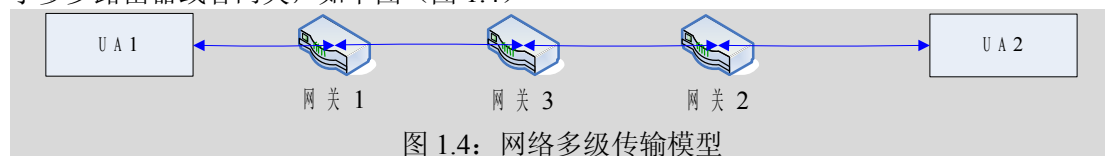


图 1.4：网络多级传输模型

每个路由器，又是同时在为不知道多少个客户服务，任务有轻重，系统有繁忙程度，因此，每个路由器的工作，实际上都是不保证的，如果中间有某一台路由器，在这一个瞬间，发生繁忙，针对你的这笔交易失败，是完全有可能的，而这一切，永远没有办法预测，只能是事后分析。

因此，如果中间的某一台路由器，如图中网关 3，在某个时刻，因为繁忙或者其他什么原因，导致传输失败，是完全有可能的。而此时，UA1、UA2 由于没有和网关 3 没有发生直接交互，是不可能知道本次传输时可能失败的。（如图 1.5）

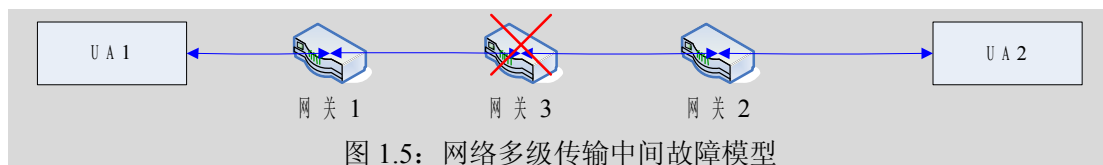


图 1.5：网络多级传输中间故障模型

这类情况还很多，因此，在数据传输这个领域，没有什么状态是恒定的，你这一秒通过这个链路传输成功，仅仅是表示这一秒链路是好的，在你得到这个结果的时候，这个信息就已经过时了，没有可参照性了。

所以，网络状况，永远只能分析，不能被观察。

商用程序员，永远不会相信一条链路，一个模块，会永远完好，也不会相信一次调用，一定会成功，他们会设置很多校验机制，会仔细检查每次调用返回的结果，随时判定服务的成功与否，并处理每一个错误的情况，书写具有较高自恢复性的程序逻辑，保证系统在任何错误下，都能自动恢复，持续服务。

这要求商用程序员，要具有极其灵活的头脑，善于逆向思维。很多时候，即使程序本身没有 bug，但某个时间点的一次传输，就是失败了，但过了就又恢复正常，在工作中，这很可能被 QA 部门作为 bug 报出来，此时的程序员要极其冷静，仔细分析，看究竟是程序 bug，还是系统本身的质量问题。

同时，商用工程系统，一般要求自恢复性非常好，绝对不能因为某一次 bug 而造成崩溃，应有足够的自我防范和恢复机制，故障出现后第一时间恢复现场，重新开启服务，使系统能继续工作。

1.3.4 小内核，大外延，工程库思维

商用系统工程，一般建议模块化开发，商用程序员可以考虑将一些成熟的代码模块做成工程程序库，以 api 层与上层业务层交互。以达到较高的重用率，降低开发成本，提升程序员的产能。

为了实现这一目的，要求商用程序员，一开始就要有建立工程库的意识，在代码实施中，始终保持高度的抽象性，遵循一个逻辑只写一次的开发理念，每一段代码的编写，均按照库代码的高标准实现编写，同时加以高强度测试，力争完成一块，成熟一块，入库一块不断提升自己的产能。

由于工程程序库为多个应用工程使用，库内包含的程序模块，库提供的 api 接口，原则上一旦确定，就不能再更改，只能做增量补充，绝对不允许删改，避免新工程的实施，导致旧工程的崩溃。

另外一方面，进入工程程序库的代码，程序员尽量仔细把握，原则上凡是与某个工程具体业务相关的代码不要入库，因为通用性不够好。库内代码尽量短小精干，每个模块功能尽量少，保持最大的应用灵活性。

笔者在开发中，个人习惯，喜欢使用一个叫 share 的目录，所有的库代码均放于此处，一个工程，不管是服务器还是客户端，甚至是 n 个工程，均直接共享该目录下的库代码，一个逻辑，一旦被库代码实现，没有特殊理由，原则上以后的应用工程不允许自行实现，必须调用库代码。

这样还有一个好处，就是在团队开发中，比较容易实现标准化开发。整个团队，基于一个成熟的库开发，配合严格的代码规范，可以有效降低 bug 率，提升团队的效率和产能。

笔者以前带团队完成的一个服务器集群，所有的 C、C++ 程序员，均是基于笔者的底层库在开发，最后，十几万行代码写下来，9 轮测试总共只测出 51 个 bug，其中属于 C 和 C++ 部分的，只有 7 个。上线后直接就是运行一年，没有任何 bug。

工程程序库的威力，大家是不是能体会到什么？

1.3.5 单笔交易失败不算失败

这是笔者长期工作的经验，在商用工程中，很多时候，设计到网络数据传输的开发，并且，一般面临的是基于公网互联网 7*24 小时不间断服务的运营需求。

在系统上线后，漫长的服务过程中，很可能会因为某一次网络的异常，或者黑客、病毒的攻击，导致服务程序本身的异常，使某笔客户交易失败。而这类故障，在实验室中很难通过模拟得到，因此，商用数据传输工程的测试，很难做到真正的 100%测试，简单说，就是不存在完美的系统。

这是正常现象，笔者建议商用工程的开发人员和 QA 测试人员，要充分认识这种不完美性。商用软件系统工程，一般强调的是智能化的异常处理，异常之后的自恢复特性，任何一次异常，造成的某一次客户服务失败，都不算 bug，只要程序能自动恢复，无资源泄漏，无崩溃，无累积性逻辑错误，系统就是成功的。

这在性能上看，也很合理，一台服务器，可能同时为几万甚至几十万客户服务，一笔或几笔服务失败，在比例上看，其实并不大，系统长期运行过程中，某个时间点可能出现性能瓶颈，但只要自动恢复，大多数时间都能正常服务，系统就是成功的。

因此，商用数据传输工程开发，有个很重要的思维，请各位读者体会：“**单笔交易失败不算失败！**”

1.4 数据传输各个角色的开发思路

在笔者通常所做的商用数据传输系统中，虽然工程千变万化，但是其中还是有规律可循的。任何一个网络工程，总离不开服务器和客户端这两个角色，两个角色的开发，各有侧重点。这里提出来，供大家参考。

一般说来服务器开发以稳定为主导思想，开发思路偏保守，尽量使用成熟技术，回避新技术的使用。而客户端的开发，由于和 UI 相关，使用技术偏新，且对程序的稳定性要求有一定放宽，毕竟客户端即使挂死，一般也不会有太大问题。

但近年来嵌入式系统的不断发展，嵌入式开发越来越成为重点。笔者粗略了解了一下，发现嵌入式设备的开发难点反而较高，这里也提出来，单独做一个讨论。

1.3.1 服务器的设计原则

网络服务器的开发，一般都有运营的要求，即放到电信机房，长期运行。因此，服务器的设计，对于系统稳定性，故障自恢复性，错误追溯性提出了较高要求。

以 Linux 平台开发居多，Windows 平台也占相当比例。

服务器设计必须对所有资源的使用，有专门的设计，及时做好资源回收，避免泄露。这些资源主要包括内存资源、socket 资源、线程资源等。

服务器针对内存，必须有特殊的设计，对于使用完的内存块，必须及时回收重用，避免内存碎片的产生。服务器应尽量减少对于动态内存的申请和使用。

服务器对线程的使用，不允许每个任务独立一个线程，应该设计时间片概念，以任务方式组织业务动作，达到线程重用的目的。鉴于 Linux 服务器的特性，一个服务器的并发线程，建议不超过 300。

服务器内部建立完善的资源跟踪机制，保证资源的申请和释放，可跟踪，可追溯。并利用服务器的 log 系统做好记录。

服务器必须自带 log 系统，log 系统必须充分考虑磁盘写满，文件超限等情况，自身设计非常安全。log 系统执行静默原则，即有错误记录，没有错误，不做任何事。如无特殊要求，原则上，服务器的 log 系统保留最少 72 小时的记录，甚至更长，以方便故障分析。

在网络协议的动作规约中，原则上服务器永远是被动方，即每次动作，总是由客户端发起，这样一来避免由于网络错误导致动作发起方的挂死，二来，由于客户端通常处于内网，外网的服务器很难访问内网的设备。但这个原则在多服务器集群中不绝对。

服务器对请求的响应原则上执行“从严”的策略，层级校验，一旦发现对方身份不对，则立即中止服务。

在服务策略的选择上，服务器执行“自私”原则，即优先保证自己的安全，再谈客户服务，对于服务器而言，单个客户的某一次服务失败，不叫 bug，应及时终止本笔交易，释放资源为下一交易服务。

服务器需要自动定时显示基本的负载信息，供操作员观察服务器负载情况，即使做出负载均衡调整。

由于很多服务器处于电信托管机房，原则上服务器有两套退出系统，控制台的键盘输入，以及远程的退出机制，便于远程管控。退出时应尽量温和退出，即按顺序释放所有资源后退出，避免磁盘文件等残留物影响下次运行。

服务器原则上工作于后台，除了退出信号，一般为哑元设备，不需要客户交互，避免不必要的 bug。

由于很多服务器实际上工作于 Linux 控制台模式，原则上服务器的开发，所有输出信息应使用英文，避免使用中文，以免乱码。

由于服务器运行的平台，通常不止一路服务，服务器对 CPU 和内存等本机资源的使用，应该有礼貌，原则上使用则申请，不使用及时释放，避免影响其他服务的正常运行。

必要时可以考虑为服务器开发专用的“看门狗”程序，监控服务器的运行状态，一遍故障时重启服务器，快速恢复服务。

1.4.2 PC 客户端的开发思路

PC 平台，大概是大家最熟悉的微型计算机平台了，近年来，PC 平台随着科技的进步，性能也越来越强大，很多家用机的性能，已经超过了过去服务器的性能。

由于 PC 平台定位在家用机和办公用机，因此，在 PC 平台上运行的，绝大多数是商用数据传输中的客户端软件，因此，这里我们主要讨论客户端的开发思路。

由于目前 Windows 平台绝对的市场占有率，一般说来客户端基于 Windows 平台开发。

客户端开发以 UI 为主，以用户体验为主，可以适当考虑使用新的技术，追求一些特效，即使由此带来可能的不稳定，也可以接受。

客户端原则上不考虑长期运行能力，因此可以使用动态内存分配，当然，必须及时释放资源，不允许读写出界，但可以不考虑内存碎片的影响。

客户端由于并发任务较少，因此一般直接开线程工作，不考虑使用池技术。

客户端原则上不考虑 log 系统，由于拥有 UI 界面，可以很方便地将信息直接与用户交互。

客户端不关心负载平衡性，对于资源的使用可以适当放宽。

客户端在不考虑上述特殊设计的前提下，应该尽量灵活多变，满足快速开发目标，以响应不断变化的用户需求。

理想状态下，客户端显式分为内核和 UI 层，二者以 API 交互，这样可以实现快速换肤之类的 UI 特效。

在与服务器交互的过程中，客户端对于服务器资源的申请，应该主动发起连接，并且，客户端负责所有失败后连接恢复的事务。且请求失败后，应该能自动恢复现场，不至于崩溃。

1.4.3 嵌入式设备的开发思路

嵌入式设备在网络中的角色一般很难说，基本上既有服务器应用，也有客户端应用，甚至二者兼有。比如一个家庭的无线网关，就很难说他的多角色身份定位。

但嵌入式设备有一个天生的局限，就是系统资源偏少，比如一台 arm9 的设备，通常情况下，64M RAM，64M Flash（模拟磁盘），相对于 PC 机动辄 1G 的内存来说，显得非常少。

这就要求嵌入式设备的开发，应该比服务器的要求更加严格，对于资源的使用，更加谨慎才行。

而且，嵌入式设备的操作系统也非常复杂，有 Palm、Symbian、Windows Mobile、Linux 等，颇有点当前 Windows 出来之前，PC 平台上操作系统百花争艳的感觉。

上述种种，都给嵌入式开发，带来了很大困难。

不过万事万物，皆有规律可循，不管哪个操作系统，在 C 和 C++ 语言这个层面上，总是差不多的。商用工程开发的规律也是差不多的。因此，从开发思路讲，各种嵌入式平台，差别并不是想象那么大。

笔者在本书之中，主要使用 Arm9 平台的 Linux 2.4 内核实现开发试验，但代码的通用性均较好，有兴趣的读者可以在本书代码的基础上，略作修改，即可适应其他操作系统开发。

嵌入式设备的开发，首先要确定比服务器更加严格的边界界定。甚至要分模块设定边界。这是因为嵌入式设备的内存、CPU 资源远低于普通的 PC 机和服务器，对资源溢出或泄漏更加敏感。

原则上，嵌入式软件对内存的使用，应使用服务器级的内存池管理，且设立更加严格的边界检查，一旦内存申请失败，即放弃服务，以牺牲服务品质，确保本机安全。

嵌入式设备的内存容量较小，直接导致 C 和 C++ 程序运行空间较小，因此，原则上不允许申请超过 4M 以上的内存（特殊需求除外），且对内存的及时回收和重用提出了更高的要求，这个原则必须贯穿于软件开发的始终。

由于前述原因，且由于线程启动时，需要分配一定的栈空间（Linux 下为 1M），嵌入式系统原则上并发的线程数一般不允许超过 50 条（一般建议 < 30 条）。且每个 C 函数内部，不允许使用超过 1M 的静态数组。

嵌入式设备运行的应用相对比较单一，很多只跑有限的几路服务，且 CPU 和内存较低，因此在设计时，对于 CPU、内存的内存，可以考虑在不挂死的前提下，尽量征用，以保证较高的服务品质。

嵌入式设备基本上属于哑元运行，且没有自己独立的输入、输出设备，一般都是远程登录管理，因此，所有信息必须由 log 系统保留到虚拟磁盘上（Flash Rom），但由于磁盘容量较小，原则上保留最后 1 小时的信息，供事故分析。

嵌入式设备通常作为设备长期运行，虽然处于用户家中或办公室，但实际上是 7*24 小时工作居多，因此，嵌入式软件产品的稳定性要求，应按照服务器的标准设定。

嵌入式设备对请求的响应原则上执行“从严”的策略，层级校验，一旦发现对方身份，则立即中止服务。

在服务策略的选择上，嵌入式设备执行“自私”原则，即优先保证自己的安全，再谈客户服务，对于嵌入式设备而言，单笔服务失败，不叫 bug，应及时终止退出，释放资源为下一客户服务。

嵌入式设备通常也没有服务器的专业管理手段，用户关机一般是直接关闭电源，没有安全退出的可能性。因此嵌入式设备对于需要长期保存的数据，不允许使用任何 Cache 机制，以牺牲性能为代价，直接写入 Flash Rom 等虚拟磁盘文件中。

嵌入式设备至少要提供一种远程管理手段，有条件的话，应该提供小型的 http 服务，供用户使用浏览器登录管理。且要充分考虑到嵌入式设备很多都工作在内网，因此，尽量提供外网到内网的浏览器直接登陆管理，这需要额外配套公网中继 Relay 手段。

嵌入式设备的业务逻辑尽量以响应请求，予以回应的“傻瓜化”操作为主，尽量非智能，系统逻辑越简单越好，条件允许时，如有中心服务器的系统，可以考虑将嵌入式设备的一些计算压力，放到中心服务器代为解决。

嵌入式设备的用户相关业务，UI 层原则上和内部逻辑层分离设计，二者尽量以异步方式通信，一来可以实现换肤等操作，二来可以尽量减少程序潜在的 bug。

嵌入式平台通常自带看门狗硬件，应加以利用，必要时系统自动重启，恢复服务。

1.4.4 跨平台软件模块的开发思路

在很多时候，商用数据传输系统，要考虑各种不同的客户需求，并且还要综合成本与利润的考虑，基本上一个系统内部，既有服务器，PC 客户端，也包括嵌入式设备，一般都比较复杂。

而作为商用程序员来说，为每个平台开发一个底层传输内核，显然是一种不经济的行为。程序员也要考虑自己的产能。

这说明，商用程序员首先要有比较强的系统分析能力，能将每个系统，与平台特性相关密切的部分（如 UI），以及通用性较好的部分（如传输内核），显式地分为不同的模块，分别予以维护。其次，商用程序员有较强的跨平台开发功力，在通用性较好的部分，尽量维护同一个模块，通过条件编译等手段，实现一套库模块，多平台通用。

笔者在网上和一些朋友沟通的时候，很多朋友都在询问，如果一套代码，跨平台通用，则这个代码必然是通用性较高的代码，那么通用代码效率低时众所周知的，如何保证效率？

笔者却认为这不是一个问题，跨平台开发，本身已经决定了，程序不可能调用太多平台相关特性，很多算法和数据结构需要自己内部实现。而在实现过程中，必然会针对我们的需求做特定优化，其效率很多时候反而高于调用平台提供的通用性系统调用。这在后面的模块说明中有详细介绍。

由于服务器的标准高于客户端，而嵌入式在服务器基础上做了更加严格的界定，因此，一般跨平台模块的开发要求参照嵌入式设备和服务器的标准执行。

所有的模块，内部不允许使用静态数组，使用动态内存分配，以保证有限的栈资源调用，以及保证边界可调，方便各个平台条件编译。

基于商业化数据传输的需求，跨平台传输库至少包含内存池、线程池、log 日志系统、基本队列、基本传输等关键模块。

数据传输内核提供通路，但不保证传输，传输内核不负责校验数据的正确性，传输品质保证由上层应用模块负责。

第 2 章 基础知识

2.1 内存的理解

- 2.1.1 32 位操作系统的内存分配
- 2.1.2 C/C++语言对内存的使用
- 2.1.3 内存----bug 之源

2.2 并行运算

- 2.2.1 时间片
- 2.2.2 进程和线程
- 2.2.3 同步和异步
- 2.2.4 礼貌的释放时间片资源
- 2.2.5 跨线程通信
- 2.2.6 跨进程通信
- 2.2.7 网络，并行运算的世界

2.3 “锁”的使用

- 2.3.1 为什么要使用锁
- 2.3.2 使用锁容易犯什么错误
- 2.3.3 “行为锁”和“资源锁”
- 2.3.4 单写多读锁
- 2.3.5 不可重入锁
- 2.3.6 用锁的最高境界，不用

2.4 “池”的深刻含义

- 2.4.1 “池”的由来
- 2.4.2 “池”的使用

2.5 跨平台、跨语言开发基础

- 2.5.1 C/C++跨平台开发基础
- 2.5.2 dll 和 so
- 2.5.3 api 和 npi
- 2.5.4 服务无处不在

2.6 Debug 的重要性

- 2.6.1 在数据传输领域，你亲眼看到的都不是真的
- 2.6.2 如何看到--万事从 Debug 开始
- 2.6.3 Debug 的原则
- 2.6.4 如何分析数据

2.7 性能统计的重要性

- 2.7.1 我们需要统计哪些信息
- 2.7.2 基本的统计方法
- 2.7.3 随机数的产生

2.8 队列无处不在

- 2.8.1 数据结构在数据传输中的应用分析
- 2.8.2 我们需要哪几种队列形式

2.9 不要求全责备

第 2 章 基础知识

商用工程开发，应对的客户需求千变万化，实现的手段也多种多样，因此，在实际工作中，需要涉及的知识面很广，也很深。一般读者很难通过看一本或几本书，获得完善的知识储备，因此，本章将就本书可能涉及的知识，做一个统揽，帮助大家理解后续章节。

当然，笔者无意写出一本标准的大学教科书，因此，在本书中涉及的知识，将自始至终贯穿需求为导向和实用主义的方针，针对需求谈技术。无关的部分被刻意忽略，因此，建议读者如果希望对本书知识有更进一步的了解，还是请阅读相关的专业书籍。

2.1 内存的理解

在 C 和 C++ 语言开发中，指针、内存，一直是学习的重点。因为 C 语言作为一种偏底层的中低级语言，提供了大量的内存直接操作的方法，这一方面使程序的灵活度最大化，同时，也为 bug 埋下了很多隐患。

因此，无论是不是做商用数据传输工程，C 和 C++ 程序员首先要对内存有一个清晰的理解。

2.1.1 32 位操作系统的内存分配

32 位操作系统，包括 Windows 和 Linux，其实都支持 4G 内存的连续访问。甚至 arm9 这类嵌入式的操作系统，由于 CPU 也是 32 位的，因此，开发时也支持 4G 内存地址的理论访问能力。

在 32 位操作系统中，通常把内存分为两个 2G 的空间，每个应用程序在运行时，就是每个进程，最大可以使用 2G 的私有内存，(0x00000000~0x80000000)。即理论上支持如下的大数组：

```
char szBuffer[2*1024*1024*1024];
```

当然，由于实际运行时，程序还有代码段，临时变量段，动态内存申请等，实际上是不可能用到上述那么大的数组的。

至于高端的 2G 内存地址，(0x80000001~0xFFFFFFFF)，操作系统一般内部保留使用，即供操作系统内核代码使用。在 Windows 和 Linux 平台上，一些动态链接库（Windows 的 dll，Linux 的 so），以及 ocx 控件等，由于是跨进程服务的，因此一般也在高 2G 内存空间运行。如下图

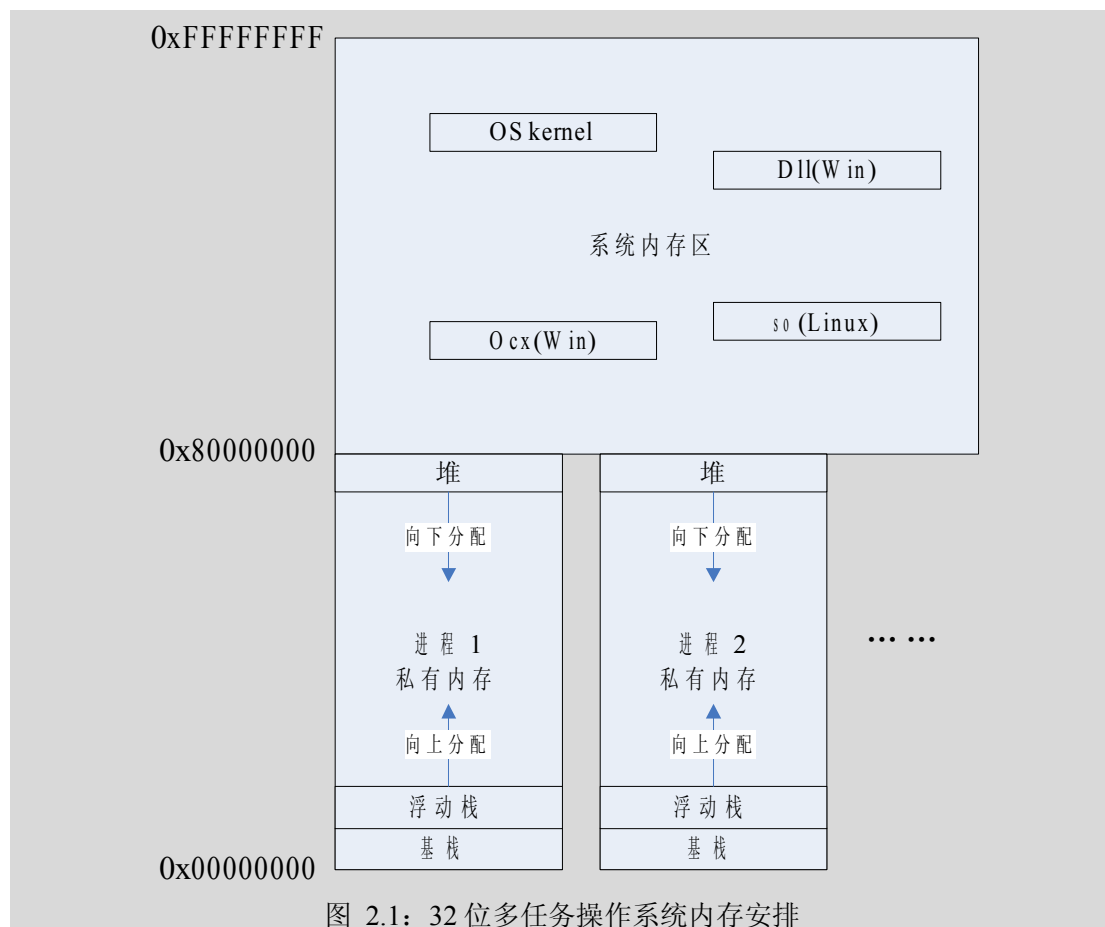


图 2.1: 32 位多任务操作系统内存安排

大家可以看到，每个进程都能看到自己的 2G 内存，以及系统的 2G 内存，这些地址对它都是有意义的。但不同进程之间，是无法彼此看到对方的。

这说明，同样是 0x40000000 这个单元，进程 1 看到的值可能是 0x00，进程二看到的，可能是 0xFF，但肯定不是同一个内存单元。

当然，作为操作系统，在底层肯定做了更多的工作，比如磁盘上的虚拟内存交换，不同内存块的动态映射等等，但对于我们应用软件开发人员来说，那些并不重要，最重要是我们能理解上面的图，能理解不同进程间私有内存空间的含义。

这里面最直接的举例就是 Windows 下的钩子程序。

每个 Windows 进程都有自己的消息队列，但某些时候，进程 2 需要了解另外一个进程 1 的消息，Windows 提供了这个机制。

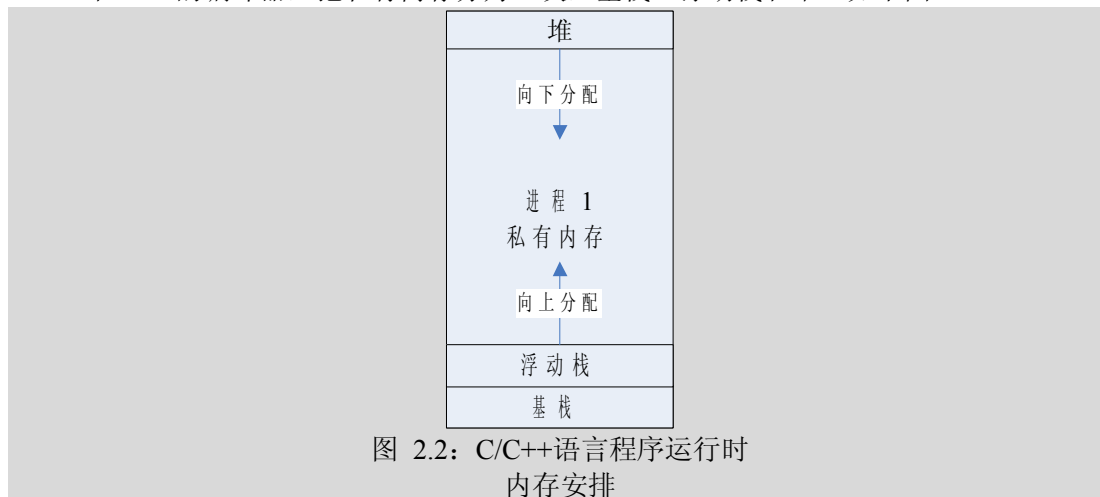
但这样有一个问题。由于进程 2 和进程 1 分属不同的内存空间，那运行在进程 1 空间的钩子程序，如何把钩到的消息，传送到进程 2。

通常的做法，就是做一小段 dll，进程 1 的钩子把信息传送到位于上 2G 空间的 dll 里，然后再通过 dll，传送到进程 2 的应用程序模块进行处理。利用上层的共享内存区，做一次中转传递。

2.1.2 C/C++语言对内存的使用

C 和 C++语言，在内存管理这一块，其实没有太大的分别。操作的都是低 2G 的内存地址空间。

C 和 C++的编译器，把私有内存分为 3 块，基栈、浮动栈和堆。如下图



1、基栈：也叫静态存储区，这是编译器在编译期间，就已经固定下来必须要使用的内存，如程序的代码段，静态变量，全局变量、`const` 常量等。程序一调入内存，至少就要占这么多空间。有个简单的不太准确的算法，`exe` 文件有多大，差不多基栈就有多大。

2、浮动栈，很多书上，就叫“栈”，就是程序开始运行，随着函数，对象的一段段执行，函数内部变量，对象的内部成员变量，开始动态占用内存，浮动栈一般都有生命周期，函数结束，或者对象析构，其对应的浮动栈空间就拆除了。这部分内容，总是变来变去，内存占用也不固定，因此叫浮动栈。

3、堆，C 语言和 C++语言，都支持动态内存申请，即程序在运行期，可以自由申请内存，这部分内存，就是在堆空间申请的。堆位于 2G 的最顶端，自上向下分配，这是避免和浮动栈混到一起，不好管理。我们 `malloc` 和 `new`，都是从堆空间申请的内存，`new` 比 `malloc` 多了对象的支持，可以自动调用构造函数。另外，这里也可以看出，`new` 出来的对象，其成员变量位于堆里面。

C 和 C++语言编程，如果要想熟练使用指针，必须对上述内存的分配原则有较为深刻的了解，程序员开发时，应随时关注变量位于哪个内存区域。

举个例子：

```
void Func(void)
{
    char i=0;
    char* pBuffer=(char*)malloc(10);
    //...
}
```

这个函数如果运行，其中 `Func` 的代码，位于基栈，`i` 和 `pBuffer` 这两个变量，位于浮动栈。而 `pBuffer` 指向的由 `malloc` 分配的内存区，则位于堆内。

在内存理解上，最著名的例子就是线程的启动时的参数传递。

一个函数，启动一个线程，很多时候需要向线程传参。但线程是异步启动的，即很可能启动函数已经退出了，而线程函数都还没有正式开始运行。因此，绝对不能使用启动函数的内部变量给线程传参。

道理很简单，函数的内部变量在浮动栈，当函数退出时，浮动栈自动拆除，里面的数据已经宣告失效，准备给别的函数运行使用了。

当线程启动时，按照给出的参数指针去查询变量，实际上是在读一块无效内存区域，程序会因此而崩溃。

正解是启动函数 `malloc` 一块内存区域，将需要传递的参数填入，将指针传入线程，线程函数收到后使用，最后线程退出时，`free` 释放。（C++的 `new` 和 `delete` 也可以）。

这实际上是利用堆内存的持久性，来度过这个空隙时间，规避浮动栈的不确定性。

这也是笔者在开发时，唯一一处不遵守“谁分配、谁释放”原则的特例。笔者称之为“远堆传参”。

举个例子：这是笔者常用的 `socket` 监听模块，当 `accept` 到一个新的 `socket` 后，通常需要启动一个新的线程为新 `socket` 服务。这里使用了远堆传参。

```
//这个结构体就是参数表
typedef struct _CListen_ListenAcceptTask_Param_
{
    Linux_Win_SOCKET m_nSocket;
    //其他参量... ..
}SCListenAcceptTaskParam;
//习惯性写法，设置结构体后，立即声明结构体的尺寸，为后续 malloc 提供方便
const ULONG SCListenAcceptTaskParamSize=sizeof(SCListenAcceptTaskParam);
//这里接收到连接请求，申请参数区域，将关键信息带入参数区域，帮助后续线程工作。
bool CListen::ListenTaskCallback(void* pCallParam,int& nStatus)
{
    //正常的函数逻辑... ..
    //假定 s 是 accept 到的 socket，需要传入后续线程工作
    //在此准备一块参数区域，从远堆上申请
    SCListenAcceptTaskParam* pParam=(SCListenAcceptTaskParam*)
        malloc(SCListenAcceptTaskParamSize);
    //给参数区域赋值
    pParam->m_nSocket=s;
    //此处启动线程，将 pParam 传递给线程... ..
    //正常的函数逻辑... ..
}
//这是线程函数，负责处理上文 accept 到的 socket
bool CListen::ListenAcceptTask(void* pCallParam,int& nStatus)
{
    //第一句话就是强制指针类型转换，获得外界传入的参数区域
    SCListenAcceptTaskParam* pParam=
        (SCListenAcceptTaskParam*)pCallParam;
    //正常的函数逻辑... ..
    //退出前，必须要做的工作，确保资源不被泄露
    close(pParam->m_nSocket); //关闭 socket
    free(pCallParam); // free 传入的参数区域
    //... ..
}
```

2.1.3 内存----bug 之源

在几乎所有的 C 和 C++相关的设计书籍中，都会提到内存，指针的使用，同时，也警告大家，无规则的滥用内存和指针，会导致大量的 bug。

根据笔者的经验，这确实是事实。在笔者过去遇到的 bug 中，几乎 70% 以上都是属于内存和指针错误。因此，笔者建议，一个 C 或 C++ 的程序员，应该对内存的使用保持高度的敏感性和警惕性，谨慎地使用内存资源。

在使用内存的时候，最容易出现的 bug，无非是读出界、写出界、野指针、资源泄漏、内存碎片等几种。这里我们说明一下：

1、读出界，这通常出现在一次拷贝动作中，由于源地址空间小于目的地址空间，而拷贝的长度又是按照目的地址空间长度设计的，就会导致读出界。严格的说，读出界问题一般不严重，VC++ 的 release 模式甚至都不报错，仅仅 debug 模式会有一个警告，不过，出于严谨的工程设计目的，应该予以避免。

2、写出界，这往往也是出现在一次拷贝动作中，与读出界相反，这往往是源地址大于目的地址，拷贝的长度按照源长度给出时出现，尤其在字符串类不定长拷贝中，特别容易出现。写出界很严重，通常直接引发程序崩溃，因此需要高度警惕。

3、野指针是指没有经过初始化的指针，由于 C 和 C++ 程序对浮动栈的重用性，内存单元在使用完毕后，通常会保留最后一次的使用结果，这对后来被分配使用该内存的程序来说，不是好事，如果不进行及时的初始化动作，指针指向的地址不可知，其操作结果也不可不知，极易造成程序崩溃。不仅仅是指针，各种野变量也可能造成严重后果，因此，C 和 C++ 程序员应该保持赋初值的习惯，任何变量，一经声明，立即赋初值。

4、内存资源泄漏，这其实是典型的二元动作问题，即一个操作，需要两次操作才完整。如内存的申请和释放，对象的建立和销毁，加锁和解锁等。如果一个程序，不断申请内存，而忘了释放，哪怕泄漏的内存再小，系统的内存空间也会在将来某个时刻耗尽，程序崩溃，这类错误非常严重，在 7*24 小时运行的商用工程中尤其敏感，因此，程序员要保持高度警惕。

5、内存碎片的产生，其实是与 C 和 C++ 无关，应该属于操作系统内存管理算法的问题。但在 C 和 C++，由于很多时候我们需要直接操作内存，这个问题表现尤为严重。

例如一个程序，申请了一块 128 字节的内存，又申请了一块 256 字节的内存，根据操作系统的内存堆分配原则，这两块内存存在堆中是顺序排列的，如图：2.3



图：2.3

之后，它释放了 128 字节这个内存块，然后它需要申请一块 192 字节的内存，由于前面的 128 字节空区放不下，于是又到 256 字节的内存块后面去分配了，如图：2.4



图：2.4

如果这种情况持续恶化，总是不断申请新的大空间，而释放的小空间又无法重复使用，最后内存就会变成图：2.5 的样子。



图：2.5

图 2.5 当然是一种模拟的理想状态，仅用于说明原理。不过如图所示，内存被隔离成多个小内存块，每个空区都不大，如果这个时候我们需要申请一块超过 128 字节的内存，系统虽然有很多内存空着，但是无法分配，总是返回失败。

程序运行到这里，虽然不会崩溃，但由于新内存申请总是失败，服务无法继续，这对于 7*24 小时运行的服务程序来说，已经可以算作崩溃了。

内存碎片问题非常麻烦，首先它是操作系统的问题，编译器无法检查，其次，从程序字面上看不出什么问题，还有就是其出现具有随机性，并且通常在运行很久后出现，一旦重启又一切正常，无法 debug。

因此，这种 bug 危害极大，不易暴露，不易检查，不易修正，商用程序员，需要高度警惕这类错误。

这些问题，在后文的 C 和 C++ 无错化程序设计方法中，笔者都会给出详细的解决方案。这里不再赘述。

2.2 并行运算

很多商用工程，一般需要多个网络角色的协同工作，这就隐含了多台计算机的并行计算，以及单台计算机内部，多个线程和进程之间的并行计算。因此，商用工程一般也可以理解为商用并行计算工程。

2.2.1 时间片

说到并行计算，尤其是单台计算机的并行计算，一定要先建立时间片的概念。

我们现在所用的，不管是 Windows 还是 Linux，一般都称为多任务操作系统。即系统允许并行的运行多个应用程序。

操作系统，一般是按照一定策略，定期给每个活动的进程，执行其内部程序的机会。并且每次只执行一小段时间，然后操作系统利用中断强行退出执行，将当前程序信息压栈，然后开始执行下一个进程的一小段程序。

通过这样不断快速的循环切换，每个程序都获得执行，在用户看来，感觉到很多程序都在平行的执行，这就模拟了并行计算。

当然，新的多核 CPU，以及超线程 CPU，内部就有超过 1 个的 CPU 执行体，运行时，就不是模拟了，而是真的有两个以上的程序在被执行。

当然在我们程序员看来，只需要理解程序是被操作系统片段执行的，每个片段就是一个时间片，就足够了。

既然是片段执行，程序员就必须理解，在自己的程序运行时，不是独一无二的，我们看似很顺畅的工作，其实是由一个个的执行片段构成的，我们眼中相邻的两条语句，甚至同一个语句中，两个不同的运算符之间，都有可能插入其他线程或进程的动作。

这说明两点：

第一，并行系统下，我们看到的程序效率，其实不是真实的效率，仅仅这台计算机，在目前部署的服务和运行情况下，以及目前系统赋予我们的优先级下，我们执行的效率。这个效率不是线性的，无法作为计算参考。

第二，我们不能想当然地认为，我们的动作不会被打断，很可能在相邻两个动作之间，已经有其他线程做过一些事情了，这些事情包括，可能一个磁盘文件，原来有，现在已经没有了，或者内容被改变了，原来有的一个共享内存区，不见了。等等。

时间片的概念，可以用来部分解释本书开始时的一句话：“在数据传输领域，你亲眼看见的，都不是真的”。

前一句程序看见的事物，下一句再看，可能会发生变化，前一句程序检测好的状态，下一句程序使用时可能不再完好，因此，如果我们有一件很重要的工作要做，要访问某个

资源，或者依赖某些条件做事，我们需要以某种特定的形式，宣告我们的动作不要被系统打断。这就是下文锁的由来。

2.2.2 进程和线程

每当一台计算机，无论哪种操作系统，将一段应用程序调入内存，开始执行，其实就是开始启动了一个进程。

操作系统启动进程，一般都会做配置运行表，分配一块私有内存空间，读入二进制程序，地址修订，分配时间片，跳转到代码开始处执行等工作。多任务操作系统，允许同时启动多个进程。

因此，进程，就是应用程序被操作系统调入内存之后，分配和合理的资源，并给予足够的时间片，切实开始运行的程序生命体。

线程则是微软公司在设计 Windows 操作系统时，认为启动一个进程的成本很高，因此建立的一个新概念。线程是附属于进程的，没有自己的私有内存空间，与父进程共享内存空间，但有独立的时间片，可以独立运行。线程启动成本很低。

事实上，大家可以认为 Windows 下，所谓的进程，在分配完私有内存空间后，就是启动了一个线程来开始从 main 函数运行。这个线程每个进程都有，我们称之为主线程。但它本质上还是线程。

Linux 下的情况稍稍复杂一点，基于 Unix 的设计思维，其实原本没有线程的概念。它内部有进程和子进程的区别。子进程基本思路其实和线程一样，也是为了节约资源而设立的。Linux 下的线程概念，更像是子进程的伪装。

但这些都并不重要，我们需要理解的是，在多任务操作系统中，可能会有很多个程序流程，在并行进行，同步工作，这是最重要的。

出于跨平台考虑，在本书中给出的示例代码库，一律以线程方式编程。

线程开发，有很多禁忌，这里简单说明，后文会有详细论证：

1、线程不能开得太快，即不能一个程序，在一个循环中快速启动线程，Windows 和 Linux 平台都有这个禁忌，否则，很可能开出死线程。即这个线程系统已经认为存在了，但没有获得时间片，实际上永远不会运行。一般建议，线程启动的间隔>250ms。

2、线程**永远不能**在外部被 kill。线程也是函数，为了实现功能，可能会 malloc 资源，可能会加锁，这些都是二元动作，即“进入-执行-退出”逻辑，如果在执行期被外部 kill 掉，则由于退出逻辑没有执行，会导致资源泄漏。原则上是设置一定条件，让线程函数自行返回。

3、线程的总数有限制，Linux 下一般一个进程为 300 条左右，Windows 可以多一些，但一般建议也不要太多，可以参考 Linux 的限额。

4、线程是后启动的，即一个函数开辟线程，很可能这个函数已经退出了，但线程还没有获得第一次可运行的时间片，因此，如前文所述，给线程传参，需要使用“远堆传参”。

2.2.3 同步和异步

既然前文我们已经理解了，多任务操作系统，其实很多任务是并行操作的，这就说明，这是一个异步的世界，大家各干各的。

异步带来最直接的好处，就是高效率。由于各个任务都在高频切换，CPU 本身不会有什么等待，一个任务发现条件不满足，可以马上把时间片释放出来，其他任务快速跟进。最高效率地利用计算资源。

但这就带来一个问题，因为实际工程中，很多业务都是有先后顺序的，需要先做一件事，再做另外一件事。这就涉及到同步问题。

这有点费解，因为我们最开始学习写程序，写个程序，从头执行到尾，其实本来就是同步操作。现在由于异步的出现，我们原来同步的事情，不再同步，缺被迫需要做很多事情来重新把程序变得同步。

因此，在多任务并行处理环境中，同步和异步，是最难控制的一件事情。

通常的同步方式，就是等待，当一个线程发现自己要工作的条件不满足时，就进入一个循环等待中，直到别的线程完成工作，自己执行条件满足时，再继续下面的工作。

这里有个结论，**多任务环境所有的同步等待动作，一定是以牺牲效率为代价的。**

在很多时候，同步又被称为“串行”动作，异步被称为“并行”动作。这很好理解，同步，先做完一件事，再做另外一件事，每次只做一件事。异步，大家一起来，呵呵。

同步和异步转换最直接的例子，就是线程的生死控制。这在下文跨线程通信中，有详细论述。

在讨论同步和异步问题时，有个话题不能不讨论一下，就是哪种计算适合同步计算，哪种适合异步计算。

一般说来，我们的计算任务分为 CPU 密集型计算和 IO 密集型计算。

1、数学计算，图形计算，C 和 C++语言基本语句涉及到的计算，如循环语句等，这类计算操作一般主要是 CPU 来完成，除非业务需要，否则一般不与 IO 口有什么密切联系。

这类计算任务，一般有个特点，即计算的顺序非常重要，后面的计算依赖前面的计算结果。因此，一般建议使用同步计算模型。原因很简单，如果我们把一个顺序执行的程序段落，强行拆分成多个线程，但是大多数后续任务由于需要前导任务的结果，而陷入等待，因此没有意义。

2、网络编程，即访问网卡的程序，磁盘访问，串口、并口访问这类接口相关的计算任务，一般称之为 IO 密集型计算。

这类计算任务，由于 CPU 和外设之间本来就是异步执行，如果我们每一时刻只允许单一任务访问外设，则由于等待网卡的数据传输等动作，使 CPU 陷入大量的空置等待中，显得很浪费。因此，一般建议使用异步模型，即多任务，多线程并发访问，利用争用使外设的利用率最大化。

因此，举例来说，比如我们要计算 100 万的阶乘，建议使用同步模型，即整个计算任务放在同一线程中完成。而我们设计一个 http 服务器，要应对多个客户的网络访问，则建议使用异步模型。

当然，上述原则不绝对，仅仅是一般的通用思维，工程实战时，还需要根据客户需求说明的计算细节，平台特征，细分任务模型，仔细挑选计算模型。

2.2.4 礼貌的释放时间片资源

在笔者刚刚开始使用线程的时候，遇到的第一个线程问题是挂死，原因就是前文所述的没有“远堆传参”导致。第二个问题，就是 CPU 使用率居高不下的问题。

原因很简单，我们通常的程序段落，一般里面都有一个循环，不断处理一些事务，直到完成工作，最后退出。

Windows 程序最能说明这个情况，以纯 C 写过 Windows 程序的读者大概都知道，WinMain 一旦启动，CreateWindows 完毕，直接就是进入一个死循环，叫消息循环，不断从自己的消息队列中提取消息处理，以此来达到事件响应的目的。直到收到一个 WM_CLOSE 之类的退出消息，跳出循环，WinMain 函数返回，进程结束。

因此，绝大多数程序，在运行时，其实就是一个大的死循环在运行。

在 DOS 等单任务操作系统编程时，大家可能习惯于把死循环写成如下形式：

```
while(1)
{
    //Do something... ..
}
```

这是正确的，因为单任务操作系统，每个任务独占所有资源，没有任何问题。但在多任务操作系统下，上述情况有所改变。

前文已经说过多任务操作系统，需要不断切换时间片来达到并行的目的，但操作系统如何决策何时切换时间片最合适呢？

一般说来，32 位 CPU 都有一个硬件的时钟计数器，当指定时钟周期到达，会发出一路中断，操作系统截获后，表示当前线程任务已经执行了足够长的时间，应该切换一下，让其他任务有机会得到执行。

不过这样显然有个问题，如果每个任务都把自己的时间片用完再释放，这一来没必要，二来操作系统自己的操作就没有足够的时间去执行了。

因此，一般多任务操作系统有个约定，即应用程序应该配合一下操作系统的时间片切换，如果一段程序，处于某种等待状态，如等待网卡数据到来等，就 **Sleep** 一下，这相当于执行了一次空操作，表示目前自己暂时可以不使用时间片，操作系统立即切换其他任务运行，这样系统显得很和谐。

应该说，无论是 Windows 应用还是 Linux 应用，绝大多数任务都是遵循这一原则的，并且，绝大多数时间片都是没有用完，即被任务主动通过 **Sleep** 释放掉。因此，我们应用程序员，在开发中也应该遵循这一原则。

因此，上文的死循环，可以写成如下形式：

```
while(1)
{
    //Do something... ..
    Sleep(1);          //强制释放时间片，降低 CPU 占用率。
}
```

另外，请读者注意，不仅仅是死循环，程序中一些较大的循环，比如上万个单元的数组的遍历查找算法，建议也考虑中间定期插入一些 **Sleep**，降低 CPU 占用率。

```
int i=0;
int nCount=0;
for(i=0;i<10000;i++)
{
    //Do something... ..
    nCount++;
    if(1000<=nCount)    //每执行一定次数，睡眠一次，1000 可以根据情况调整
    {
        nCount=0;        //计数器归零
        Sleep(1);        //强制释放时间片，降低 CPU 占用率。
    }
}
```

顺便再说一句，不仅仅是 C 和 C++，其他拥有线程功能的语言，也可能需要考虑这个问题。请程序员关注以合理的 **Sleep**，释放时间片资源。

2.2.5 跨线程通信

通常在工程设计中，我们各个任务之间，并不是彼此无关的，不同的任务线程，通常需要各种通信，协调彼此的动作。因此，线程间如何通信，是我们并行计算的一个很重要的话题。

简单说来，跨线程通信，无非是通信的双方线程，均访问同一个内存区域，读一个变量的值，因此，只要做好了锁保护，跨线程通信和直接读取变量没有差别。读者可以使用后文介绍的“资源锁”来解决这个问题。

我们这里举出线程的生死控制作为通信的示例，请大家分析。

我们知道，线程不可以从外面直接杀死。只有等每个线程完成工作之后，自行退出函数，线程自动结束。

这里面就带来一个问题，如果程序退出时，主线程期待退出，但还有其他子线程在工作，此时如何协调一起退出？

要知道，如果主线程不等不看，自行退出的话，系统会立即拆除这个进程的私有内存空间，此时所有子线程访问的内存单元会全部被系统宣告作废，程序立即以崩溃结束。

不管是 Windows 和 Linux 系统，都提供了大量的信号量，临界区等机制，来负责各个线程间的通讯，以达到退出时的协同退出问题。

不过出于跨平台开发起见，笔者一般是使用多线程安全变量来完成。这个具体的代码释义，在后文有详细论述，在这里，我们主要探讨的是如何控制。

如果要完美地控制线程的起停，我们需要做两件事情，一个是主线程有条件通知所有线程，自己要退出了。另一个是所有线程退出后，要有一个信号，通知主线程退出完毕，可以继续。

笔者通常的控制方法，是用两个变量来控制线程

1、一个是 bool 变量，叫做 bThreadContinue，顾名思义，这是所有线程可以持续运行的标志。因为在商用工程中，通常有很多守候线程，即该线程不断做死循环，等待某个事件的发生，该变量控制死循环是否继续。这个变量，用作主线程来掌控所有线程的生死。

2、一个是 int 变量，叫做 nThreadCount，也很容易理解，所有线程的计数器。每当一个新的线程启动，这个计数器+1，线程退出，计数器-1，当计数器为 0 时，显然，所有线程已经退出。

这里是程序示例。使用的是 Windows 操作系统和 VC++6.0

其中，MBOOL 和 MINT 就是多线程安全的变量，即内部的所有运算都是加锁的操作，是线程安全的。后文有专门论述，大家此处可以先不必理会。

大家可以看到构造函数 CMyClass::CMyClass() 中，开启线程的方式，先累加线程计数器 nThreadCount，然后调用 AfxBeginThread 启动线程。

线程函数写在类里面的时候，一定要写成 static 属性，这是因为所有 C++ 的类成员函数，都有个隐含的属性 this，指向本对象的指针。

而线程函数属于系统调用，系统是不理解什么对象的，也就无法理解这个 this 的含义，因此，必须使用 C++ 的静态成员函数来书写线程函数，也就是在函数参数表中强制取消这个 this 变量，才能被系统正确回调。

```
class CMyClass
{
public:
    CMyClass();
    ~ CMyClass();
protected:
    static UINT MyThread(LPVOID pParam);    //线程函数，必须是静态的
    MBOOL m_bThreadContinue;                //线程循环标志
```

```

    MUINT m_nThreadCount;          //线程计数器
};
CMyClass::CMyClass()
{
    //控制变量赋值
    m_bThreadContinue=TRUE;
    m_nThreadCount=0;
    int i=0;
    //开启个线程
    for(i=0;i<100;i++)
    {
        m_nThreadCount++;          //一定要在开启线程前累加线程计数器
        AfxBeginThread(MyThread,this); //开启线程，以本对象指针为参数
    }
}
CMyClass::~CMyClass()
{
    m_bThreadContinue=FALSE;      //通知线程关闭
    while(m_nThreadCount!=0) {Sleep(1);} //等待所有线程退出
    //此时退出，线程已经全部安全退出!!!
}
//线程函数
UINT CMyClass::MyThread(LPVOID pParam)
{
    CMyClass* pThis=(CMyClass*)pParam; //得到本对象指针
    while(pThis->m_bThreadContinue) //此处以控制变量的值作为循环继续依据
    {
        //循环体
        //Do something ...
        Sleep(1); //礼貌地释放时间片
    }
    pThis->m_nThreadCount --; //线程计数器减 1，通知主线程自己退出
    return 0;
}

```

2.2.6 跨进程通信

在很多商用数据传输工程中，一个需求，通常是一台或几台计算机上运行的多个服务共同完成的。而这些服务，分属不同的进程，甚至分属不同的计算机，如果要想实现彼此协调工作，共同完成服务，那么，如何进行跨进程通讯，就是非常必要的。

其实各个操作系统，都提供了很方便的跨进程通信手段，如信号量，位于高 2G 内存的共享内存区等等。最起码，磁盘文件是各个进程都看的见的，也可以算作跨进程通信的一种手段。

但商用工程不太一样，特别是很多基于分布式的系统工程，由于通信的双方，很可能是不同的服务，为了部署的方便，这些服务，可能在一台计算机上，也可能多台，此时使用单机模式的共享内存，信号量等机制就不合适了。因此，为了保证最大的部署灵活性，笔者建议：“统一使用基于 TCP/IP 协议的 socket 传输作为进程间通信手段”。

socket 通信是网络接口，天生就是为了不同两台计算机通信而存在的，而 TCP/IP 协议，又允许本机内部通信，因此具有极大灵活性。

这最直接的例子，就是笔者近期的一个电子白板的工程。工程需求为电子白板按照会议室方式运行，每个客户端都有写的权利，写出的笔画，能被所有用户看到，并且，新加入的客户，不会丢失笔画。

这就要求电子白板客户端有一个会议室数据中心，用于汇总所有人的笔画，并保留所有客户端绘制的笔画，会先同步到数据中心，然后广播给所有人。原始需求里面有个特殊的说明，这个数据中心其实也是一个客户了，就是会议室的主持人，它具有最高的绘制权利。

好，我们来分析一下，看来客户端分为两类，一类是主持人，一类是一般客户，主持人一个要支持本地的绘制和显示，同时还是会议室服务器，接收所有人的数据，放到本地的数据中心，同时广播给每个人。这么做，当然没有问题。

但笔者分析时，想到了一个可能性，如果有一天这个系统公网运营，数据中心和主持人分离了怎么办？这完全有可能。

经过思考，笔者决定把系统分解为数据中心和普通客户功能两个部分，二者之间使用 socket 通信，主持人模块，其实是一个数据中心和一个普通客户端的结合体而已。

结果很喜剧，项目还没做完，笔者就接到通知，确实有客户的环境，就是要求主持人和数据中心分离，如果笔者按常规思路，主持人模块内置数据中心，则这个程序改动量就很大。但由于笔者的模块是用第二种方式划分，并且使用 socket 作为通信接口，一切都没有问题，直接把数据中心模块部署到一台服务器上运行就 OK 了。代码修改量为 0。

2.2.7 网络，并行运算的世界

前面说了那么多，无非是给各位读者建立一个基本印象，网络，其实就是并行运算的世界。

这很好理解，各个系统模块之间，其实没有必然的联系，彼此都是在异步工作，齐头并进地完成自己的业务。

这说明，如果本书的读者希望实现商用分布式的开发，首先需要对编程的观念做一个调整。将以前单一任务的，串行的，同步的开发理念，变成多任务的，并发的，异步的开发理念。

我们知道，异步最大的好处，就是高效率，使资源的利用率最大化，因此很多网络服务的优化，都是通过将同步动作异步化来完成的。比如以前一台服务器完成的工作，拆分到几台服务器上，一起协同工作，完成任务，系统性能立即提升好几倍，甚至几十倍。

这里提示一点，如果读者以后面临类似提升服务器性能的需求，可以首先考虑拆分业务，将每笔业务都拆分成为单一服务，然后通过异步运算，提升整体服务器集群的性能和容量。

但异步最大的难处，其实就是将异步的动作，由于业务的需求，重新变成同步，这个成本很高，涉及到大量的等待动作，一般建议是尽量从设计方面规避。

在网络中，异步转同步有一个很大的隐患，需要大家注意。就是由于等待造成网络延时叠加。

网络通信的双方，很多时候，是并不知道对方以及连接线路的真实情况的，因此，在通信动作的同步上，有很多断言，推论，揣测活动，这些行为由于没有显式的判断条件，因此一般都是以超时作为失败标志。

比如某个程序想做个动作，发出信号后，等待一段时间，如果对方不回应，就不再死等了，直接宣告失败退出，避免自己因为等待被挂死。

但这时候可能会带来一个问题，有些时候，由于业务需要，A 计算机想 B 计算机发出一个请求，期待得到这个回复，这就是一个带超时的等待。而 B 计算机发现自己手里资料

不齐，需要从 C 计算机那里调阅一些资料，于是，向 C 又发出了一个请求，开始新一轮的延时等待。C 又有可能向 D 询问……，于是，一个巨大的延时等待链诞生了。如下图

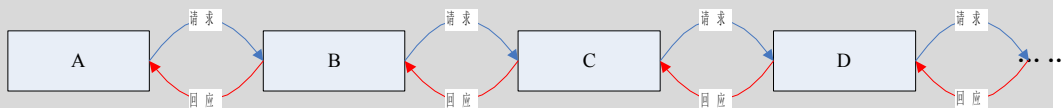


图 2.6

这带来一个问题，A 的超时时间需设置为多少秒，才能判定请求是失败了？

由于各级请求都有延时等待，理论上讲，A 要等待的时间，是所有请求等待时间之和，但我们知道，B 的请求是 B 临时决策发起的，A 并不知情，C 的也一样。

从网络编程的角度，这个问题没有任何解决方案，因为作为模块开发的程序员，很难站在系统设计的高度来统筹安排超时时间。另外，这个问题就算出了，也不算他程序的 bug。

这只能从系统思维的角度来缓解问题。一般说来，A 的超时时间尽量长一点，大家可能关注到，不管网络速度有多快，哪怕所有的动作都是在几百个毫秒内完成，但 IE 浏览器预设的超时时间一般在 30 秒以上，其实就是在试图规避这种延时等待链。

另外就是商用系统设计的时候，就要避免这类延时的叠加，使服务器集群尽量扁平化，避免太深的请求递归层数。

这类潜在的隐患还很多，商用网络工程，是并行运算的世界，和传统的程序设计有很大差别，读者以后在实施类似的工程时，需要打破固有思维，仔细分析，尽量规避隐患，才有可能做出真正合用的商用工程。

2.3 “锁”的使用

前文已经说明，在商用数据传输中，面临的是一个并行的开发环境，同步和异步处理是软件研发的技术核心。而对于多任务操作系统而言，异步转同步最常用的工具，就是“锁”。

“锁”有很多形式，如临界区，信号量，等等。本书为了理解方便，统一称为“锁”。

2.3.1 为什么要使用锁

锁的由来，起源于 CPU 对内存的操作。我们知道，不管 CPU 如何进步，16 位到 32 位，再到现在的 64 位，其实主要说的是两点，一个是 int 这个变量类型，即整型数，代表的位数，这表明了计算机一次运算中能处理的最大数字。一个是 CPU 的地址寄存器，由于位数的增加，使可以连续编址，同时访问到的内存空间在大幅度扩大。

但不管怎么变化，我们发现，计算机的内存，其实还是以字节（Byte）在计数，即 8 位二进制，是一个内存单元。CPU 每个时钟周期，至少能通过总线访问到 1 个字节。

我们以 32 位操作系统举例，当 CPU 要做一次最简单的整数加法计算时，会一次从内存读入 4 个字节的数据，这是一个 32 位整数的位宽，然后做加法运算，然后将数字存回内存。

但这就带来一个问题，由于是多任务环境并行运算，这个加法计算可能会被打断，比如一个计算，我们把 0x000FFFF 做加 1 的计算，结果应该是 0x00010000。

但实际运行时，当计算机准备将 4 字节的数据写回内存时，只来得及写了低位两个字节，就被切换了。这个时候，整数的数值就变成了旧的高 16 位+新的低 16 位，0x00000000，这显然是个错误的数字。

如果很不幸，这时候有另外一个线程，也需要访问这个数据，正好切进来读取，就读到了一个明显非法的结果，而依赖这个数据的后续计算，就都不对了，严重的，如果这个单元是一个指针的话，就可能导致程序直接崩溃。如下图。

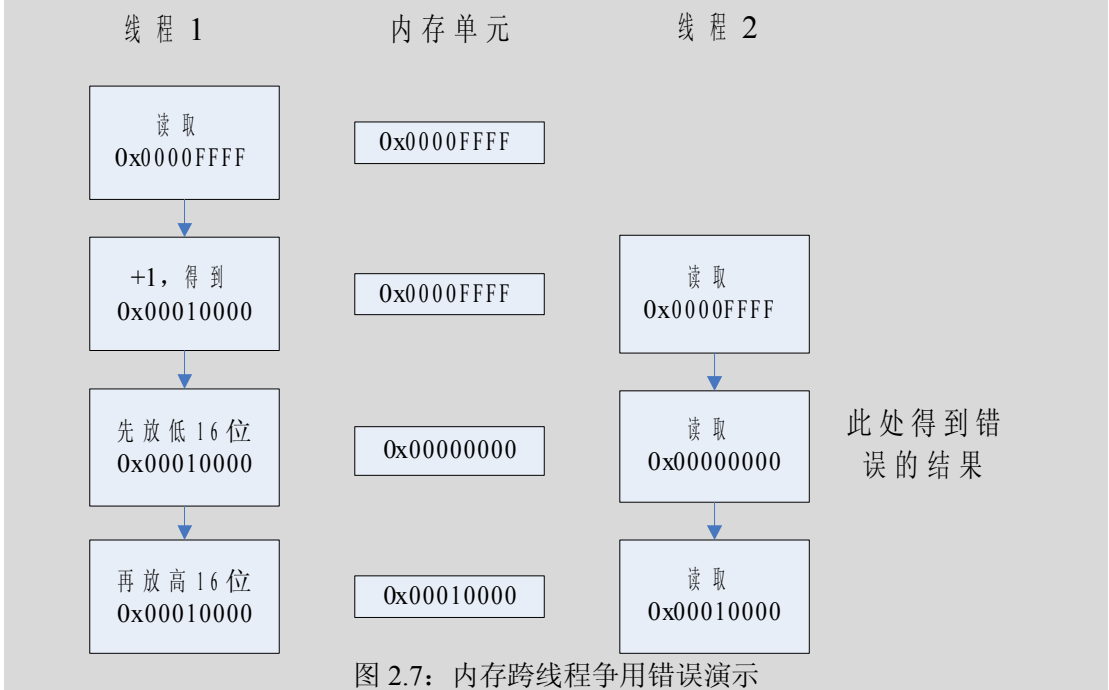


图 2.7: 内存跨线程争用错误演示

这种错误非常可怕，出问题的往往是另外一个线程，不是制造事故的第一责任人，因此很难从错误点来跟踪源头，这类错误，一般都应该从一开始就尽量避免。

多任务操作系统针对这个问题，提出的解决方案就是加锁。加锁事实上是一种宣告，告知操作系统和 CPU，本线程的下面几个动作不能被打断，直到解锁为止。

因此，当两个线程都按照“加锁-访问-解锁”这个顺序做事时，如果第一个线程加锁后，第二个线程试图访问加锁的资源，则被系统悬挂等待，直到锁解除后，才可以继续执行，这样就能确保读到正确的数据。如图 2.8

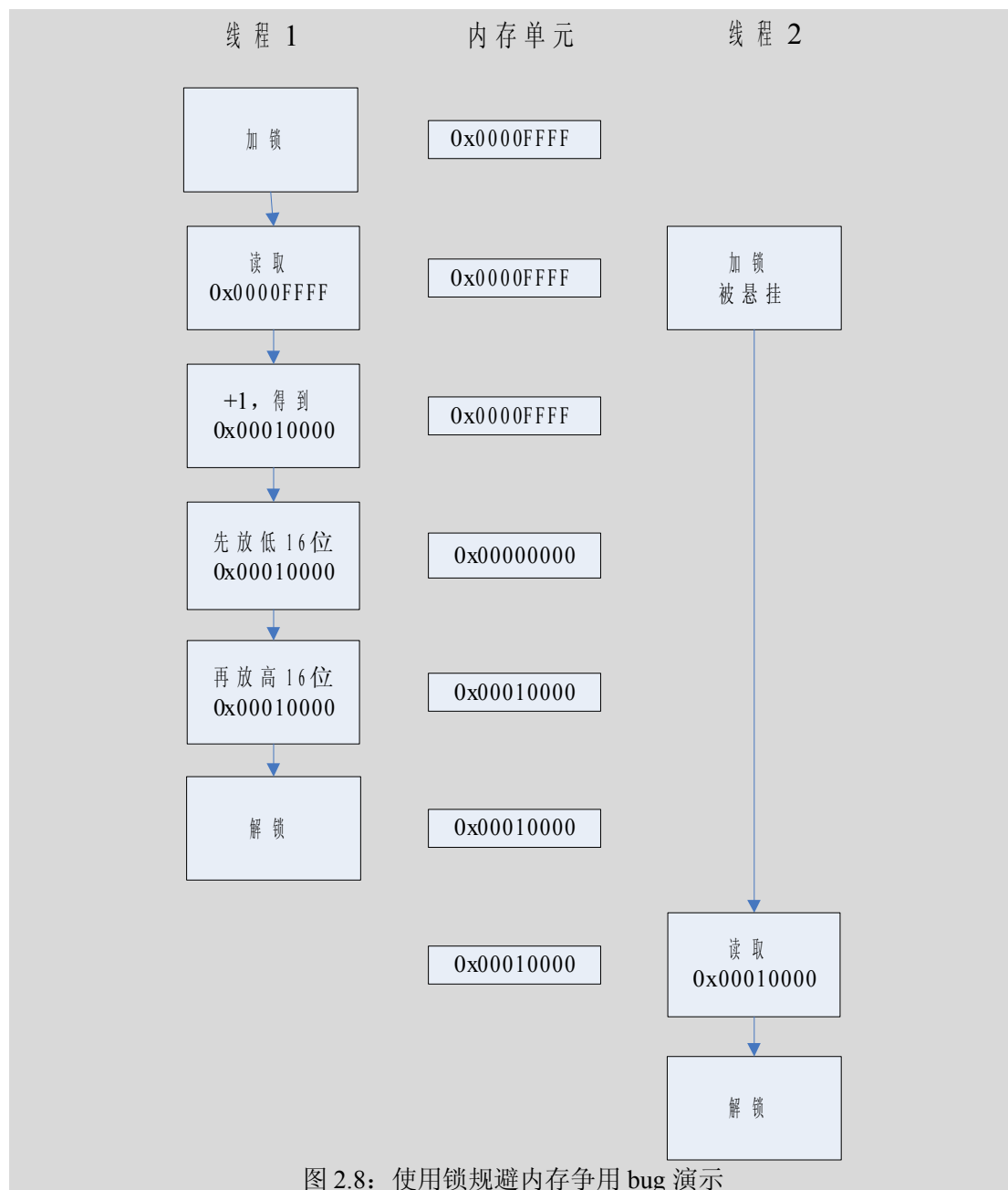


图 2.8: 使用锁规避内存争用 bug 演示

由此可见，使用锁，是不同线程访问相同资源的必要条件，否则程序的运行结果将不可知。

在商用工程领域，这个“锁”的概念还可以进一步抽象，经常是我们处理一笔复杂业务，需要连续几个动作不能打断，这时候，也可以使用锁这个概念。

但这个时候，锁的形式就多种多样了，可以使一个上面提到的原子锁，也可以使一个字节的内存单元保存的状态（1 字节的操作不会被打断），也可以是一个数据库的锁定动作，甚至，可以专门设计一台锁服务器，协调服务器集群中，各个业务服务器之间的同步协调。

2.3.2 使用锁容易犯什么错误

2.3.2.1 中途跳出导致忘了解锁

很多程序员，在使用锁的过程中，都犯过错误，包括笔者自己。而最常见的一种错误，就是忘了解锁。

我们知道，锁是二元动作，这和 malloc/free 的动态内存分配释放有点类似。这类动作一定是两两配对，缺一不可，同时也不能多做。

体现到程序设计上，很多时候，程序员可能会在某种跳出机制的时候，忘了解锁，导致下次进入锁的线程，被永远悬挂，最终，程序被挂死。

比如下面这几个例子

```
void Func1(void)
{
    m_Lock.Lock();
    //...
    if(l==n)
    {
        //...
        return; //此时 return, unlock 没有执行，锁没有解除，以后的程序会挂死
    }
    //...
    m_lock.Unlock();
}

void Func2(void)
{
    while(1)
    {
        m_Lock.Lock();
        //...
        if(l==n)
        {
            //...
            break; //在 while 循环中跳出，循环最后的 unlock 没有解除，以后程序挂死
        }
        //...
        m_lock.Unlock();
    }
}

void Func3(void)
{
    while(1)
    {
        m_Lock.Lock();
        //...
        if(l==n)
        {
            //...
            continue; //提前解除循环，循环最后的 unlock 没有解除，以后程序挂死
        }
        //...
    }
}
```

```
m_lock.Unlock();  
}  
}
```

上述几个例子，都是由于程序员的粗心，而导致中途跳出时，忘了做解锁工作，以后的程序被挂死。这类 bug 非常有害，因为发生挂死的是下面的程序，可能在本线程，也可能不在，但不管怎么查，下面的程序书写是没有问题的，不能通过跟踪的方式查出 bug。

笔者在工作时，曾经发生过为了跟踪一个锁 bug，连续加班半个月都找不到问题的情况，锁出现的问题有时候非常难以查找，程序员必须对此有很高的警惕性，尽量杜绝此类错误的发生。

提示：工程开发中，笔者有个简单的判断方法，如果一段程序经常出现一些莫名其妙的 bug，而且从程序字面上看，没有任何问题，根据经验，笔者建议开发人员应该检查锁了，一般都是因为某个资源忘了加锁，导致线程争用所致。而另一方面，如果多线程工程，正在运行时，从日志上查看，发现某一线程的日志不再增加，则很可能是因为 double lock，该线程被某个锁挂死。

2.3.2.2 Windows 检测重复加锁的漏洞

另外还有一个很容易犯的错误，实际上是 Windows 操作系统不严谨造成的。

原则上，锁的存在，就是杜绝错误的访问，因此，如果一个锁已经被锁住，另外一个试图加锁的操作一定被悬挂住。但经过试验，笔者发现 Windows 不是这样。

笔者在工程实践之中，很多次都发现，如果在同一个线程中，对一个锁资源重复加锁（Double Lock），Windows 很可能无视这类明显的错误，继续执行，但也不出错。

比如这段代码：

```
void Func2(void)  
{  
    m_Lock.Lock();          //从调用逻辑上，这是重复加锁，应该挂死  
    //...  
    m_Lock.Unlock();  
}  
void Func1(void)  
{  
    m_Lock.Lock();          //此处已经加过一次锁了  
    Func2();  
    m_Lock.Unlock();  
}
```

大家可以看到，上述代码明显是错误的，由于程序员笔误，导致两次加锁，但 Windows 系统从来不报错，而且执行顺利，程序逻辑正确。

由于笔者经常写跨平台程序，上述代码代入到 Linux 下用 gcc 编译执行时，会被立即挂死。可见，在这点上，两个平台有很大差异性。偏偏这类错误笔者本人常犯，弄得非常痛苦。

对于这个问题，一般也没有太好的解决办法，只能每写好一个模块之后，在 vc 和 gcc 下分别编译运行一遍，当两种测试环境都正确运行，才认为这段代码已经通过。

另外，这个问题还可以考虑使用下文资源锁的特殊写法来做一定的规避。

2.3.3 “行为锁”和“资源锁”

在笔者的开发生涯中，上述关于“锁”的错误曾经频频发生，而且几乎每次错误，都付出了巨大的 debug 代价，严重影响了开发效率，也导致了无数次加班和项目延迟，笔者就思考，难道真的没有解决方案吗？

笔者发现，其实大多数时候，程序员有个特点，即喜欢按照自己的开发需要来使用锁，举个例子，一段程序要对一个资源发生一些动作了，程序员就书写“加锁，做动作，解锁”的流程。这样，锁的使用和程序发起一个动作或者一个行为相关，笔者称之为“行为锁”。

但这样有个恶果，我们对锁的使用遍及程序的每一个角落，只要程序段落中需要安全访问某个资源，就会有一次加锁和解锁动作。而这么多处使用点，只要有一处笔误或者遗漏，就是 bug。这个过程，显然值得改善。

笔者一直强调一个观念：“**用一个习惯，解决整整一个方面的 bug**”。

笔者发现，除了极个别的特例，绝大多数锁的使用，其实是和资源相关，一个资源，一旦有跨线程访问需求，就需要加锁。这和谁来访问，怎么访问，访问什么，以及访问多少次无关。

因此笔者认为，正确的做法，应该是把锁和资源绑定到一起，利用 C++ 的类封装特性，将资源私有化，同时所有的公有，全部内部实现加锁访问，这样，不管谁来访问，只要通过该类的公有方法访问，就相当于强制加锁。

由于这个锁与资源密切相关，笔者称之为“资源锁”。

这个方法很有效，笔者自从 2000 左右发现这个方法之后，基本上就没有出过什么忘了解锁之类的错误。

这里举一个例子：

下面的类声明中，对 `m_szBuffer` 的所有访问，仅能通过 `CopyIn` 和 `CopyOut` 的公有方法来访问，由于这两个方法函数内置强制加锁动作，因此，所有的访问均是线程安全的。而这种加锁模式，也就是典型的“资源锁”。

```
class CBuffer
{
public:
    CBuffer();
    ~CBuffer();
public:
    void CopyOut(char* szDestBuffer,int nDestBufferSize)
    {
        m_Lock.Lock(); //此处强制加锁
        {
            //...
        }
        m_Lock.Unlock();
    }
    void CopyIn(char* szData,int nDataLength)
    {
        m_Lock.Lock(); //此处强制加锁
        {
            //...
        }
        m_Lock.Unlock();
    }
}
```

```
private:
    char m_szBuffer[100];    //被保护的资源
    CMutexLock m_Lock;      //这是锁对象，后文有详细介绍，这里不赘述。
};
```

不过，这种写法，也有一个问题，就是在一个类中，部分函数要考虑锁，部分不考虑，这在设计时没有问题，但在程序员实做的时候，很容易把脑子弄混，写错。

尤其是在某些逻辑中，一个公有方法，需要调用另外一个公有方法，由于所有公有方法加锁，很容易就出现了上文所说的重复加锁（Double Lock）的错误。

所以，工程实践的时候，笔者比较喜欢另外一种写法：即把一个线程安全类显式分为两块逻辑，一块专门管功能实现，另外一块专门管安全加锁。实际书写时，先写一个基类，实现所有功能，但是不加锁，另外再写一个聚合类，来包容基类，聚合类的公有方法与基类完全一模一样，只做加锁动作。

这种写法虽然最麻烦，但也确实最安全，笔者凡是用这种写法的代码，迄今为止没有出过一例锁 bug。

还是上例，我们看换一种写法的结果。

```
class Cbuffer    //这是实现功能的基类
{
public:
    Cbuffer();
    ~Cbuffer();
public:
    //请注意这两个公有方法函数的构型，这里函数实现具体的功能逻辑
    void CopyOut(char* szDestBuffer,int nDestBufferSize);
    void CopyIn(char* szData,int nDataLength);
private:
    char m_szBuffer[100];    //被保护的资源
    //此处已经没有锁了，因为本类只实现功能，不管安全性。
};

class CbufferWithLock    //锁安全包容类，命名习惯：“基类名+WithLock”
{
public:
    CbufferWithLock();
    ~CbufferWithLock();
public:
    //请注意，公有函数的构型完全一模一样。
    //公有函数内部就是加锁，调用，完全不实现任何功能
    void CopyOut(char* szDestBuffer,int nDestBufferSize)
    {
        m_Lock.Lock();    //此处强制加锁
        {
            m_Buffer.CopyOut(szDestBuffer,nDestBufferSize);
        }
        m_Lock.Unlock();
    }
    void CopyIn(char* szData,int nDataLength)
    {
        m_Lock.Lock();    //此处强制加锁
        {
            m_Buffer.CopyIn(szData,nDataLength);
        }
    }
};
```



```

    }
    m_Lock.Unlock();
}
private:
    CBuffer m_Buffer;        //需要保护的基类
    CMutexLock m_Lock;       //这是锁对象，挪到这里了
};

```

2.3.4 单写多读锁

前文我们说明同步和异步关系时，有一句话，“多任务环境所有的同步等待动作，一定是以牺牲效率为代价的”。

这说明，虽然我们用锁解决了线程间对资源的并发争用安全性，但是，随之带来的，就是程序效率大幅度的降低。也许，原来每秒几万次吞吐量的服务器，在完善了锁安全后，性能降低到每秒只有几百次的吞吐量。

这个问题看似无解，因为商用服务器，首保稳定性和安全性，无法稳定运行，什么都免谈。但在笔者的开发生涯中，笔者发现，通过对业务动作的细分，还是有可能提升性能的。

细心的读者可能观察到，前文举的用锁的理由，是对内存单元出现写操作时发生的，但如果是读操作呢，由于内存单元没有任何变化，因此，跨线程的读操作，天生就是安全的，无需加锁。

这就带来一个话题，其实对于资源的访问，所有的读，都可以是并发的，但写动作，一定具有排他性，既不能有其他读，也不能有其他写。

这就产生了一种单写多读锁的应用工程实例，这在教科书上很少提到，属于商用工程程序员在开发中发明出来的。

单写多读锁，简单说，就是对资源的访问分为两种状态，一种是读操作，一种是写操作。由应用程序提示锁应该做哪种操作。当为读模式时，所有的写动作被悬挂，而读请求被允许通过，而写动作时，所有操作被悬挂。并且，读写切换时，有足够的状态等待，直到真正安全时，才会切换动作。

单写多读锁作为解决并行安全高效问题的利器，在很多场合都在使用，以笔者自己的工程实例来说，一个简单的内存池，在使用传统锁的时候，效率约为每秒钟 7500 次分配或释放动作，但一旦改为单写多读锁，则效率暴涨到 27500 次左右。当然，这不是线性关系，也和业务访问的密集度相关，不过至少说明单写多读锁效能很高。

单写多读锁实例，在后文有详细介绍。

2.3.5 不可重入锁

在锁的使用中，笔者还发现一个有趣的需求。

一般说来，锁的逻辑就是如果已经加锁，后续的加锁动作，就被同步悬挂，执行的线程会被挂住，不再执行后续代码。

但有的时候，我们也希望加锁的动作不要悬挂，如果发现已经加锁，就返回一个假，表示加锁没有成功，上层程序知道这个资源条件不满足，就会去做别的事情，过一段时间再来。

还有的情况是，商用工程可能会有这样的逻辑，即某种条件到来后，需要使用一个资源，但这个资源还没有初始化，于是就先初始化，再使用。不过问题来了，这个资源也许只需要初始化一次，但我们知道程序是并行的，当某一个线程初始化资源时，其他线程很

可能也切进来访问，由于初始化尚未完成，其他线程没有检测到可用资源，就可能会重复做初始化动作，造成 bug。

正确的做法是，第一个初始化的线程应该做某种标志，表示初始化正在进行，虽然没有可用资源，但是后续访问者不应该再行初始化，仅仅是本次访问失败返回即可。这就需要用到不可重入锁。

不可重入锁的逻辑通常是 Lock 的时候，如果成功锁住，则返回一个真，如果已经被别人锁住，则立即返回一个假，但线程不会被悬挂。

这个不可重入锁，在后文也有详细介绍。

2.3.6 用锁的最高境界，不用

虽然说了很多关于锁的使用方法，也强调了锁在并行运算中的重要性，但是，笔者还是提醒各位读者，锁不要滥用，用锁的最高境界，是不用。

这话应该不难理解，如果从系统设计出发，我们规避了一个资源出现线程写争用的可能性，则就不需要用锁。这要求程序员不仅仅是做程序那么简单，而必须站在系统设计的高度，去分析每一个锁，是否真的有必要存在。

锁其实是为了解决并发访问时，乱序写动作造成的数据不一致问题。细分下来，很多时候可以不用锁的。

- 1、针对一个资源所有的操作都是读的时候。
- 2、只有一个线程访问的时候，即资源只有唯一持有人。
- 3、同一时刻只有一个线程访问的时候，即串行访问模式。

既然有这么多方法，其实在系统设计时，我们可以通过精心设计，规避很多不必要的锁使用。比如：

- 1、对于很多查表运算，表在程序 Init 的时候就初始化成功，之后只读，则不需要加锁。
- 2、对于线程自己的变量，无其他人访问，不需要加锁。因此，将变量尽量私有化，可减少锁的使用。
- 3、利用事务批处理机，事务处理队列，将同步动作改为异步动作，资源的访问永远具有唯一性，则也不需要加锁。

这里面唯一值得争议的是第三点，笔者在网上和朋友沟通时，有朋友就专门提出了疑问。因为队列本身，作为一个多线程共享的交互中心，也是需要加锁的。因此，这类同步改异步动作，其实是换汤不换药，没有意义。

对此，笔者是这么看的。队列进出是要锁，但要仔细分析锁的时间效率，第三种方法，实际上是一种 move loading 的方法，通常情况下不建议采用的。只有特定情况才需要。

举个例子：

网络 IO 的效率远远低于内存运算，其风险很大。如果一个 socket 池，需要加锁，则大量的任务会被阻塞到某一个 socket 的一次失败通信后的重 try 上，如果系统设计为同步操作，则这种迟延，会蔓延波及到前级的 accept 接纳，因此，就会由于某一个客户的错误，引发全体用户的服务品质下降。

换言之，对于网络 IO 的加锁操作，其实是一个时间成本为 On 的操作，即操作时间不确定。如果照此设计，则系统性能堪忧。

而如果使用一个内存队列来做缓和，某一次操作的失败，最多导致队列中增加几条数据，但所有入队列的时候，加锁的操作的时间成本基本上是 O1，前级的服务线程能迅速脱离出来，接纳下一条请求。

从外面看来，系统性能并没有受到影响。

这实际上提升了系统的容错性，和稳定服务的品质。

因此，这种方法，可以有效降低加锁时间效率的不确定性，以恒定的加锁成本来适应变动的网络状况。

而我们设计服务器时，恰恰会在很多时候遇到这种需求，所以经常会用到这种优化方式。

正所谓世事无绝对，Move Loading，本来是商用数据传输工程严厉禁止的行为，但在此处使用，恰恰起到了正面的作用，可见，我们开发人员应该随时保持灵活的思维，善于逆向分析，正确认识“没有绝对的正确，也没有绝对的错误，只看合用不合用”这个道理。

2.4 “池”的深刻含义

“池”是高性能服务器集群中，使用得比较多的一个概念。

简单说来，“池”的含义就是，一个集合，内部有一些资源，这些资源在这个集合表现的应用中，具有完全对等性，即每个资源都具有相同的服务能力和服务品质。因此，当用户来申请使用时，池可以随机挑选出一个资源来为当前请求服务。

“池”其实就是对于一个与业务相关的结构、算法或动作在高度抽象化之后的资源集合。

“池”内的资源，可能拥有内存等静态资源，等待用户申请使用，这类资源池笔者通常称之为“被动池”，而有些“池”则其内部资源本身就是线程运行体，拥有时间片资源，通常在守候某一事件，并根据事件主动发起动作，这类资源池笔者通常称之为“主动池”。

2.4.1 “池”的由来

当计算机网络刚开始发展的时候，那个时候关于网络服务的理论还不多，人们在设计一个网络传输时，通常是用“1 服务器—1 客户机”模型，即通常说的“C/S 模型”。在这种模型下，一个服务器为一个客户端服务，客户端独占全部服务器资源，享受最优质的服务。

很多单机大型游戏的架构有这样 1-1 的服务模型，后台的游戏引擎提供游戏相关的资源服务，通过脚本语言编写的游戏进度脚本实现 UI 控制。这样做的好处主要是利用 C/S 理论，强行割裂引擎和具体游戏内容的关系，使一套引擎可以做出多套游戏，降低游戏的开发成本。

同时，由于 1-1 服务，其效率和 C/S 逻辑混合在一起编程几乎一样，可以保证游戏的执行效果。

但在大多数应用环境下，这样做显然太浪费服务器的资源，并且也不符合 C/S 模型的初衷，即利用服务器强大的计算能力，同时为多个客户端服务，降低客户端的计算压力，并通过服务器资源重用，来降低整个系统的平均成本。

因此，各位读者在实际中，看到更多的 C/S 网络模型，应该是“1 服务器—n 客户机”的一对多模型。比如，网站服务器，就是同时应对多个浏览器客户端的访问。

当然，从这个例子，大家也可以看出一个道理，B/S 模型，其实也是 C/S 模型的一种。是在 C/S 模型的协议栈上，进一步增加了一层 html 解释体系的工作模型。

当然，这时候有一个问题就出现了，就是服务器的效率问题。从某种意义上讲，C/S 模型这么多年的开发史，其实可以看做是服务器效率的优化史。目的很简单，就是让一台服务器能承载更多的客户端服务请求，尽可能地降低系统成本。

做过一点系统优化的读者可能能感觉到，通常情况下，通用系统，很难做出很好的优化效果，专有系统，则一般很容易做出优化，这是因为通用系统，考虑的问题太多，要应对各种客户需求，而优化，通常是根据某种关键业务需求，刻意突出系统某一方面的性能，因此，通用系统很难做出优化。

这个最明显的例子是图形用户界面 GUI 的优化，大家都知道 Windows 的 Win32api 有专用的 GUI 函数，但效率很低，低到无法满足游戏的高速动画需求，微软公司被迫写了个 DirectX 游戏引擎，专门做游戏类的优化。这主要就是因为 GUI 为了通用，很那优化，而 DirectX 是游戏专用，可以做针对性优化。

服务器在优化过程中，也出现了这个问题，程序员发现，如果把服务器看做一个整体，则很难优化，因此，程序员开始试图从业务细分的角度，来拆分服务器的各个功能，把功能归类，然后实施优化。

比如说，客户端连接上来的 socket 连接，通常都是客户端会主动发起请求报文，然后服务器响应，这在业务上就有相似性，那么我们是不是可以让负责 listen 的线程，只管 accept 动作，不要去处理具体的连接业务，而将所有 socket，归到一个特定模块中处理，并根据这个业务特征，实施优化呢？

答案是当然可以，这其实已经形成了“池”的概念。即通过业务归类，将相同资源组织到一起，实施针对性优化。这个池，由于是守候方式工作，自己拥有时间片，应该是“主动池”。

另外，服务器内部，为了避免内存泄漏和内存碎片，通常会自行准备一个内存“池”，来解决动态内存分配的问题，这个内存“池”，拥有内存这类静态资源，是被动接受外部用户调用，因此，通常称为“被动池”。

2.4.2 “池”的使用

池其实是一个抽象概念，并且池本身就是为了优化而产生的一个概念，因此，池的实施没有一定之规。没有绝对好的“池”算法，也没有绝对差的，只有合用不合用的区别。

通常资源的组织形式多种多样，常用的数据结构如数组、链表、树，都可以用来做池的组织形式。这些结构的选择，一般根据业务特性来判断。

池有个很重要的基本特性，就是具有平行扩容性，即池中间可以任意加入和删除资源，而池本身的服务不会崩溃，最多影响一点服务品质。

池在商用工程实施过程中，通常在后期优化阶段时才被引入，即先用普通方式解决问题，待性能测试发现瓶颈后，考虑使用池技术来缓解。这也是商用工程“**先解决有无问题，再解决好坏问题**”的体现。

笔者以前曾经犯过一些过度设计的错误，请大家关注，有很多时候，笔者从实施的开始，就开始引入池的概念，然后凭自己的想象，开始优化池，而这个时候，工程尚未完成，也没有性能参数供参考，因此最后效果一般都不好，基本上属于无用功。所以后来笔者吸取教训，一般都是到了白盒测试阶段才引入池优化。

池优化一般优化的是概率、比例，得到的结果是某个百分比的提升，而不是 0 变成 1。这是由于池的先天特性决定的。

前文已经说过，池是将资源高度归纳化后，提高使用效率，因此，池能提升服务品质的比例，能提成服务成功的概率，但对于绝对化的问题，池的效果有限。比如一个支持 5000 用户的服务器通过池技术，可望提升到 8000，但要时支持 100000，则无论用什么池都无法达到，只能更换平台，重构系统才可能实现。因此请大家注意，**池不是万能的**。

在笔者做系统设计和分析的时候，通常在最后的性能优化考虑时，有一句话，“**毕竟，概率是站在我们这边的**”，其实说的就是在最后万不得已的时候，我们还有池优化这条路。

在 C/C++ 语言商用实战中，常见的池有内存池、线程池、任务池、主动连接管理池和被动连接管理池等。这在后文有较为详细的介绍。

而在商用工程的具体业务中，池技术也几乎无处不在，哪怕其开发人员并没有将之称为“池”。一个服务器集群，一个分布式数据库，其实都可以看做池技术的具体表现，因为

它们都是通过资源归类，针对性优化来实现高性能和高可靠性服务。而通常所说的动态负载均衡，其实就是服务器池的优化策略。

2.5 跨平台、跨语言开发基础

前文说过，商用工程设计，永远是至上而下的设计模式，即根据需求，先选择平台，再选择语言，再决定解决方案。这说明，商用程序员经常会遇到跨平台、跨语言开发。

当然，这给程序员带来一些困扰，需要掌握的知识面拓宽，学习难度提升，同时，代码重用度不高，重复劳动增加，不利于提升程序员的生产效率。

这些困扰，往往会造成项目实施成本增加，降低商业项目的盈利率。因此，探索一条跨平台、跨语言的开发之路，就成为商用工程开发必要的探讨项目。

这里，笔者根据多年的研发实战，给出一点经验建议。

2.5.1 C/C++跨平台开发基础

通常情况下，C 和 C++语言本身就是跨平台的，因为其语言特性本身是独立于平台之外的。但商用开发时，有时候为了保证效率，大量使用了操作系统平台相关的特性。因此破坏了软件的跨平台特性。

根据笔者的经验，主要在以下几个方面存在特例：

1、线程相关的调用：前文我们提过，Windows 下的进程更像线程，而 Linux 下的线程更像进程，二者实现机制不一样，因此，在这两个平台下开发，线程的起停、调用的代码差别很大，跨平台开发时，需要专门定义。

2、时钟相关的调用：Windows 系统设计时，由于偏重用户 UI 体验，因此做了大量方便用户的内部功能，这导致 Windows 系统显得比较臃肿，基本上不具备高精度计时特性，其计时精度通常在毫秒级。而 Linux 作为工程型系统，计时精度很高，基本上在纳秒级，基本可以算作实时操作系统，这部分调用差别很大，必须分别处理。

3、锁相关调用，在 Windows 和 Linux 下编程习惯差别很大，原因和线程差不多，主要是实现机制不同导致，因此，有必要统一锁的调用。特别提醒读者关注的，就是前文所述的，Windows 对于同一线程的 Double Lock，不报错的缺陷，在 Linux 下则会导致程序挂死，因此在锁调用时必须严密关注。

4、socket 调用，虽然 Windows 和 Linux 都宣称支持标准的 socket 库，但二者的实现机制差别还是很大，导致调用模式和返回值都有很大差别，因此，必须有一个统一的调用环境。

基于上述原因，在商用工程开发时，一般需要考虑使用 C 和 C++的条件编译语句，对上述关键点差异性做模糊化处理。

Windows 下一般使用 VC 开发，Linux 下则使用 gcc 开发，笔者一般的习惯是同一段代码，在 VC 下编译运行通过后，马上利用 Linux 虚拟机，验证其 Linux 正确性。

根据笔者经验，哪怕是苹果公司的 OS 系统，由于基于 Unix 标准，其跨平台开发也可以参考本书的原则。

2.5.2 dll 和 so

跨语言开发，相对比较麻烦一点，由于本书预设读者是 C 和 C++的程序员，因此，跨语言开发有点不太方便。在商用项目实战时，通常跨语言开发时由不同程序员完成。

不过，笔者根据经验，发现诸如 PHP、Java 等脚本语言，在不同平台运行时，均可以调用动态链接库，这在 Windows 下是 dll 文件，而 Linux 下是 so 文件。对于某些已经成熟的 C 和 C++ 代码，可以考虑做成动态链接库形式，来支持上层脚本语言的调用。

笔者并不是推荐这种形式，因为这实际上是换汤不换药，没有起到跨语言开发，综合各种语言优点，降低开发成本的目的。因此建议仅仅是针对某些 C 或 C++ 语言已经实现的功能模块，可以考虑以这种方式重用，降低开发成本。

还有一种需求也可以考虑这种开发模式，就是在某些效率优先场合，可以考虑把部分关键算法由脚本语言改为 C 写，提升系统效率。

2.5.3 api 和 npi

由于有这种跨语言合作需求，因此，api 变得非常重要。

在笔者看来，商用工程，基本上都是一个栈的结构，下层相对上层为功能层，上层相对下层为业务层。逐层搭积木，而构建整个系统。

每层之间，使用 api 来挂接，api 标准恒定，且只增不删改，保证 api 的一致性，可以实现上下层模块分别更新和升级。因此，模块的可替换性，是因为 api 的存在而存在的。

根据笔者经验，各级 api 的设置，应该考虑纯 C 方式，这是因为不管 Windows 和 Linux，在动态链接库输出 C++ 类这个问题上，都有太多限制，十分不便，很多脚本语言都不支持。而纯 C 模式的函数构型输出，则能很好得被所有语言接收，这基本上可以算是 api 的标准了。即 api 的标准输出.h 文件中，需要有如下声明，且不能输出类。

```
#if defined(__cplusplus) || defined(c_plusplus) //如果是 C++编译器
extern "C" { //强制 C 方式输出
#endif
//... ..
#if defined(__cplusplus) || defined(c_plusplus)
}
#endif /*defined(__cplusplus) || defined(c_plusplus)*/
```

api 一旦确立，业务层和功能层即可做无缝切换，如图 2.9

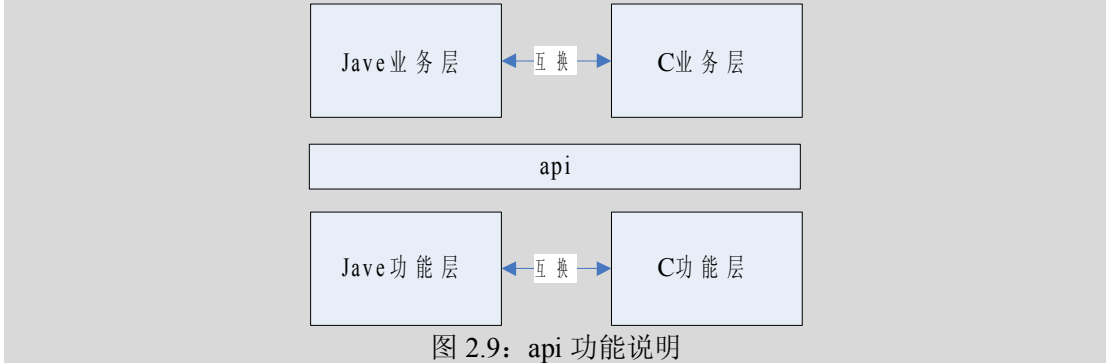


图 2.9: api 功能说明

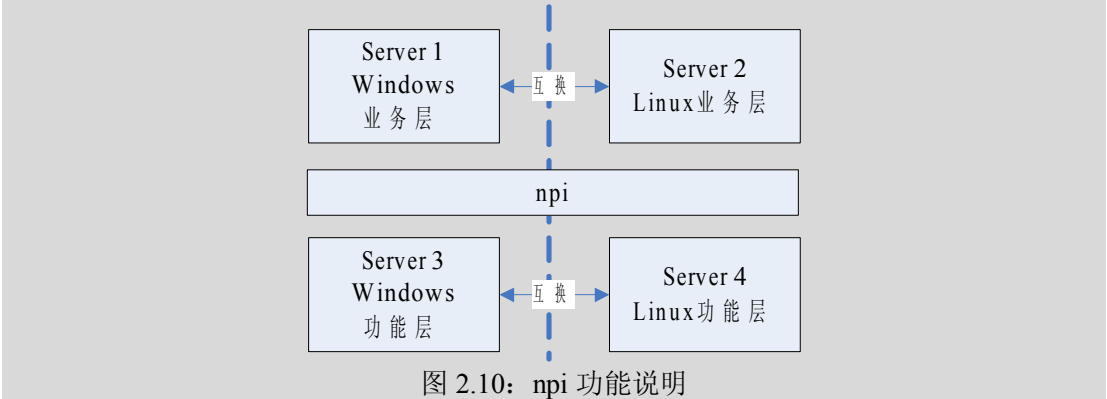
当然，有了 api 之后，npi 就很好理解了，就是基于 socket，以远程方式调用一个远端计算机上的功能。

说到 npi，很多时候，笔者喜欢用 Windows 下的 COM 组建来做比较，其实，COM 做的事情，已经很接近 npi 了，也是通过一个本地代理，直接以类、接口方式输出功能到本地运行程序，但功能实际上是通过代理自动调用远端计算机完成的。

但 COM 有个最大的缺点，就是只有 Microsoft 一家支持，跨平台性能不好。另外，以笔者的经验，COM 的效率稍微低了些。

npi 很多时候，和协议密不可分，一个功能调用，可能就是一个协议里面的一条报文，一个请求。在商用数据传输工程中，笔者更愿意把 **npi** 和 **api** 等同来看，因为二者的本质都是调用一个功能，仅仅是调用的方式不同而已。

当 **api** 进一步升华为 **npi**，则连运行平台都可以做无缝切换。如图 2.10



2.5.4 服务无处不在

由于我们建立了 **api** 和 **npi** 的概念，其实我们已经可以建立另外一个概念了，就是 **service**（服务）。

前文说过，在笔者看来，功能层对业务层而言，其实就是一个服务而已，至于这个服务在本机，还是在远端的一台服务器上，其实并不重要。

因此，**npi**，更像是服务的粘合剂，一个系统，只要明确定义各种 **npi**，将各种功能逻辑从物理实现上分隔开，理论上讲，这个系统天生就是分布式的，其任何一个模块，可以以任何一种语言，任何一种平台，实现在任何一台服务器上。

当一个服务器体系，内部全部是以服务来划分时，其分布式特性、平行扩容特性将获得最大化的方便。

因此在笔者看来，商用工程从接到客户需求，第一步设计开始，其实就是在细分功能，划分服务，这个过程贯穿商用工程开发的始终。

2.6 Debug 的重要性

很多时候，商用工程之所以开发难度和成本都比较高，主要的原因就是实验环境难以搭建，开发人员很难模拟出真实的运营环境，使调试非常困难。

但归根到底，笔者认为主要原因就是因为一个系统工程中，一切数据都是流动的，常规的调试手段，很难跟踪数据的变化，对于 **bug** 也就很不好解决。本节就是向读者展示，为什么在商用数据传输工程中，**Debug** 重于一切。

2.6.1 在数据传输领域，你亲眼看到的都不是真的

这其实是一个观念，前文已经提出过，这里再次强调。这句话的含义是，不要用习惯性思维分析问题，不要单纯用眼睛观察问题，在数据传输领域，甚至很多商用工程领域，都需要养成分析问题的习惯。这里面的含义很多：

1、数据传输业务，是流动的，是随时变化的，没有恒定不变的状态，每当我们看到一个状态，其实，这仅仅是过去某一个时间点的快照，在我们看到它时，它已经作废了，新的状态，需要重新检测。

2、IP 网络的特性，就是随机选路，一条报文，从 IP 地址 1 发送到 IP 地址 2，中间经过的路由是各级路由器根据忙闲算法随机选择的，一批报文，很可能每条报文走的路径都不一样，因此，我们不能仅仅根据前次的收发成功，就想当然推论后一次收发一定会成功。即使 100 条报文，已经成功了 99 条，最后一条，也不能推论一定就会成功。

3、不要去相信协议，TCP 号称确定传输，很多人就想当然地认为以 TCP 方式传输，就可以不做数据校验了。笔者曾经问过迈普公司做路由器的人员，如果一个路由器堆栈满了，怎么办，他们的回答是无条件抛弃，不管是不是 TCP 报文。因为路由器作为网络服务器，首先执行“自私”原则，即遇到问题首先保证自己安全，再谈客户服务。单笔失败不算失败，丢包率的存在是合理的。

4、不要去预测报文到达的时间，网络上多级路由器，各自有各自的传输队列，报文一般采用“存储转发”模型，一条报文，从出发到到达，可能经过了多次队列缓冲，异步切换，且不同报文路径各异，因此，其到达时间没有线性关系，不具备可比性。程序员唯一能控制的是，报文是按发送的先后顺序到达的。

5、互联网环境和局域网环境有很大差别，不能以局域网的测试结果来推测互联网传输的效果，必须实测。这其中，最大的差别，互联网上传输出错几乎是必然的，而局域网中则不出错是正常的。

综上所述，数据传输领域开发原则，尤其是涉及公网的项目，原则上总是比较悲观，我们理解，数据传输的每一次传输行为，其实都是一次冒险，一次测试，因此，针对每一次的传输行为，都要有详尽的故障检测和自动恢复机制，程序严禁“裸奔”。

在很多开源代码上，我们通常可以看到如下的代码段：

“send(szData,nDataSize); //发送一笔数据”

这段代码中，程序员可能由于经验所限，没有对传输调用的返回值做详尽的检测，直接想当然地认为调用会成功。对于这种写法，笔者戏称为“裸奔”。

正常的传输代码，一般每一步都带有详细的返回值校验，任何一次失败，都会导致本次传输宣告失败，同时也宣告本次 socket 失败，请上层业务层重新建立连接，重新尝试收发。

这里给大家提示一小段笔者的传输代码，大家可以看看如何实现错误管控。网络编程不是本书的重点，此处列出，仅供大家建立一个感性认识，一个商用数据传输工程的收发函数，写起来有多麻烦。

```
//非阻塞式写，从 buf 写指定长度的字符到 socket 中
//成功写出，返回写的字符数
//因某种原因，没有写出，返回 SOCKET_RW_0_NO_ERR
//错误，返回 SOCKET_RW_ERR_10054(-1)或 SOCKET_RW_ERR_OTHER(-2)
ssize_t write_all (Linux_Win_SOCKET fd, void *data, SIZE_T len)
{
    ssize_t n, m, r;
    int err;
    char *wdata = (char*)data;
    int nEWouldBlockWaitCount=EWOLDBLOCK_WAIT_MAX;
    //注意，写是非阻塞式的
    for (n = 0; n < (int)len; n += m)
    {
        //循环保证确定写够
```



```

    if(!SocketIsOK(fd))
    {
        r = SOCKET_RW_0_NO_ERR;
        return r;
    }
    m=send(fd,wdata + n,(int)(len - n),Linux_Win_SendRecvLastArg);
    if(0<m) nEWouldBlockWaitCount=EWOULDBLOCK_WAIT_MAX;
    else if (m == 0)
    {
        //写和读不一样,只要能写,就一定能写入, 如果写入为,表示现在不能写
        r = SOCKET_RW_0_NO_ERR;
        return r;
    }
    else if (Linux_Win_SocketError == m)
    {
        err=getErrno(); //本函数已经为 Linux 实现
        if(ECONNRESET==err)
        {
            //对方主动断掉链接,防火墙超时
            GetSocketError(err);
            r=SOCKET_RW_ERR_10054;
            return r;
        }
        else if ((EWOULDBLOCK==err) || (EINPROGRESS==err))
        {
            //阻塞下的超时等待逻辑
            m = 0;
            nEWouldBlockWaitCount--;
            if(0>=nEWouldBlockWaitCount) //最大超时时间
            {
                r=SOCKET_RW_ERR_OTHER;
                return r;
            }
            else Sleep(1);
        }
        else
        {
            //这是其他类错误,只能直接返回错误
            GetSocketError(err);
            r=SOCKET_RW_ERR_OTHER;
            return r;
        }
    }
}

return (ssize_t)len; //正常写完,返回写的字节数
}

```

2.6.2 如何看到--万事从 Debug 开始

由于商用工程通常面临无法模拟实际情况的开发困境，因此，其最重要的开发原则之一就是：“万事从 **Debug** 开始”。

在很多年以前，笔者第一天产生创建工程代码库的想法，写出的第一个库函数，就是一个 **Debug** 函数。原因很简单，当时笔者使用 **gcc** 开发，缺乏 IDE 联合调试环境。但大家由此也可以想象一下 **Debug** 功能对工程开发的重要性。

其实笔者的 **Debug** 函数很简单，就是一个变参函数，可以同时向控制台和文件打印需要输出的内容，但就这么一个简单的函数，几乎解决了笔者这么多年 90% 以上的 **debug** 问题。

很多程序员比较喜欢使用 **VC** 作为开发环境，一方面是 **Windows** 开发需要，另外一方面是单步跟踪等调试手段齐备，调试方便。但在商用工程开发中，笔者基本上不用，而全部以自己的 **Debug** 函数代替。由于这种 **Debug** 方式类似于使用 **printf**，所以笔者和团队成员戏称为“**print 大法**”。

原因很简单，在多任务并行计算环境下，在以数据传输为主的业务模式下，任务的切换，报文的流转都是高速进行的，任何一种单步跟踪手段都无法胜任调试工作。

当我们把某段程序停下来开始单步跟踪观察时，别的线程或网络角色经常成百上千个报文已经流转过来，由于被本次单步跟踪阻塞住，往往引发大量的网络超时，重传，一个报文被我们看清楚时，往往已经被通信的对方宣告无效。

因此，高速并行计算模型下，单步跟踪的内容，早已经失去其正确意义，没有任何参考价值。唯一的解决方案，就是把所有报文打印到日志文件里面，事后再来分析。

当然，在实际工程应用中，笔者这个 **Debug** 输出功能，已经被丰富为 **Debug** 模块，**log** 日志模块等多个系统级模块功能，其功能也不仅仅是 **Debug**，同时兼顾运行期日志输出。

而这种日志分析的 **Debug** 方法，也不仅仅在 **C** 和 **C++** 工程中使用，笔者所在的团队，在 **PHP**、**Java** 等脚本开发中，也大量使用。

2.6.3 Debug 的原则

出于尽可能暴露 **bug** 的原因，商用数据传输工程的开发通常都带有详尽的日志系统，用于输出各种错误，以及跟踪运行期重大事件。一般的使用原则如下：

1、**Debug** 一般分为 4 级，**Syslog**，**Debug**，**Debug2** 和 **Debug3**，可以分别开关。其中，**Syslog** 记录系统重大模块起停事件，所有的错误输出。**Debug** 用于业务报文跟踪，**Debug2** 和 **Debug3** 主要用于频率很高的信息输出跟踪，如收发的信令，资源池的申请释放事件等。

2、所有的模块，原则上自带 **PrintfInfo** 接口，以内部数据定义的特殊格式，**Debug** 输出数据，供 **Debug** 模块调用。

3、报文传输的跟踪，通常在接收方打印，发送方不做重复打印。

4、日志系统必须具有高安全性和高可用性，且能自适应，对于磁盘文件满等异常有自动容错机制，以简化后续各种系统调用。

2.6.4 如何分析数据

数据分析是一个很大的话题，通常根据具体业务的不同而不同，很难一概而论。不过，从商用数据传输的角度，我们抽象大多数传输行为，对于传输数据的分析，还是可以找到一定规律的。

1、数据发送的顺序不会变。通常确定的数据发送动作，不管是我们业务协议，还是系统协议，都规定有一定的 **Ack** 机制，即对传送质量有个信号回传，使发送方可以测得发送

的成功性。并且，很多收发机制都是“等停”式顺序发送，这说明，不管报文的路径如何，数据发送的顺序不会变化，这是我们在检测信令时，可以预期的。因此，如果信令发生顺序发生错乱，则程序必然有 bug。

2、信令生命期内不会消失，一条信令，从生产者产生出来，层级传递，到达每个需要通知的网络角色，最终完成使命终结，这是其生命期。一条信令，在系统设计逻辑上，没有到达最终目的地前，应该不会消失，即在每个经过的网络角色的接收端和发送端，都应该可以看到该信令的流过，如果一条信令中途消失，则可以推论程序有 bug。可能某些网络故障会导致信令阻塞，但从商用工程的开发要求，应该有足够的故障自恢复机制，信令可以迟到，但不应该消失。

3、信令生命期内不会改变，这实际上是一个系统设计的原则，一条信令可能包含多条信息，供多个网络角色使用，但每个使用者，均只能使用，而不能修改信令，否则可能会人为制造很多 bug。当这个原则一旦确定，则信令的一致性可以作为我们检测 bug 的基础。

4、网络角色透明性，这个原则是信令生命周期内不会改变原则的推论，即每个网络角色，除了信令的终结者，其他的角色均应该是透明传输，即信令在接收点和发出点应该保持一致，如果不一致，则说明传输链路中某个角色的程序逻辑一定有 bug。

5、故障通常在发送端，我们通常检测信令的地点在接收端，习惯性的，如果发现 bug，我们会在接收端检查代码。但数据传输，通常问题都出在发送端，因此，如果接收端发现问题，程序员应考虑到发送端检测 bug。

6、长时间运行的一致性，商用工程，至少有一个重要测试是必须进行的，就是长时间，大收发量，大报文量，大业务压力的风暴测试。我们开发人员通常有句话：“商用工程，没有几百万报文打底，谁也不敢说稳定！”，就是说这个原则。各种服务器，嵌入式设备，基本上都有 7*24 小时运行要求，内部对任何一个资源的不合理使用，没有及时释放，如内存、socket 等，可能在程序上没有 bug，但这类淤积，对于服务器就是不可忍受的 bug。大任务量的测试，就是专门测试这类 bug 的。通常笔者一个传输模块的测试报文量，在几百万到几亿。

7、野蛮测试的一致性，商用工程通常在公网运营，而公网环境极其复杂，实验室测试一般很难模拟。不过，通过野蛮操作，可以大致检测系统的稳定性。比如突然拔掉网线，某台服务器突然断电，又加电，等等，在进行野蛮操作时，要求所有相关网络角色，自身安全，信令不丢失，状态自恢复，即待故障恢复后，可以自动恢复正常传输状态。

8、好的程序是静默的。不管如何设计 debug 体系，基本上都有个原则，即有错误才打印日志。没有错误，程序的界面显示，基本上都是固定不动，或者循环显示某些性能信息。因此，如果一个工程写出来，虽然测试能通，功能也基本实现，但老是打印一些故障状态自恢复信息，异常状态信息，通常就是程序还有 bug，并且，bug 极有可能是信令握手出了问题。

2.7 性能统计的重要性

商用数据传输系统，以解决客户需求为目的，以实现服务为目的，这就说明，仅仅是程序写出来没有 bug，是不够的，最终能否成功完成，后期的性能测试和统计非常重要。甚至很可能后期，因为某个关键性能不达标，造成整个工程重构的现象。

2.7.1 我们需要统计哪些信息

一般说来，商用工程，最终目的是为了部署运营，因此，统计信息，不仅仅是为了 debug，更多的是作为计算基础来测算公网运营需要的带宽条件，计算机配置条件等。

根据笔者经验，我们常见的统计信息，包括性能统计和 debug 统计两方面。

1、性能统计

根据笔者的实际经验，程序设计的理论值与实测值的差别均比较大，甚至可能超过 1:10，因此，仅仅靠程序设计时预设的各项指标，对实际运营基本没有指导性，必须实测得出。

对于商用工程而言，各项指标不一定要求最高，关键是系统的平衡性必须保证，不能因为某个单项指标的优秀，影响其他指标的表现。

性能统计虽然属于后期的测试统计，在开发的时间段比较靠后，但仍然与前期系统分析密不可分。系统分析和设计时，应明确给出各个关键参量的理论值，并提出方案保障措施。而后期的性能统计，基本上是在验证设计目标是否达到。

所有的性能统计，均包含峰值统计和常规业务统计，在笔者工作中，一般喜欢测试满负荷收发报文和 1 秒 1 条报文两种情况，这基本能模拟出大多数网络情况。

主要测试的指标如下：

1、各个链路的传输速率，一般折算成 kbps，即多少 bit（位）每秒，这是电信运营商常用的带宽单位。比如一秒传送 50k 字节，应该乘以 8，即 400kbps。

2、报文交换率，即每个链路每秒钟收发的报文总数，这主要从业务层来计算每秒能执行的用户服务交易数，进而推算用户服务能力。

3、各个服务器的 CPU、内存占用情况，这主要用来计算各个服务器的最低配置。一般说来，常规业务压力下，各个服务器的 CPU 占用均较低，峰值压力下，能有效抢占 CPU 资源，将 CPU 与内存尽量“吃光”，以系统最大性能来应对极限压力。这种变化是观察的重点。

4、各个服务器的最大并发量，这是应对峰值客户服务的关键，可以通过单位时间内完成的用户交易数来计算，测试时，应该有多台客户端模拟峰值运行情况。

2、Debug 统计

另外，养成随时统计的习惯，对于我们的开发中，实现程序自侦测，自动检查 bug 非常有帮助，在后文的 C 和 C++语言无错化程序设计中，笔者将会向大家展示使用统计方法来降低非可视性 bug 的技术。

简单说来，内存块泄漏，socket 泄露，指针泄露，几乎是所有 C 和 C++程序员都遇到的必然 bug。

而这类 bug，通常由于隐藏在程序内部，很难通过肉眼观察分析，并且，由于其运行随机性，在测试时一般都是偶然出现，并且故障对运行环境具有很强的依赖性，很难重现。如某个客户发出的某个报文，某两个功能执行的碰撞等小概率事件，才会引发 bug，这导致程序员很难分析，也无从解决。

对于这类 bug，笔者也是经过了多次痛苦的历程，才想到利用统计方式，由程序内部的实现资源自统计，泄露自报警的机制，来杜绝这类 bug。经过多年的商用系统实用，证明效果很好，在本书的后续章节，将详细向各位读者解说这种方法。

2.7.2 基本的统计方法

日常的开发中，统计方法有很多，但很多书籍，均详细讲解软件开发后期，以“黑盒测试”的角度，如何测试性能。但这类方法，有个最大的问题，就是仅仅能确定程序有某些方面的性能问题，但一般不能显式描述性能瓶颈的具体位置，这很难帮助程序员迅速定位和解决问题。

因此，笔者在本书中强调，性能测试是“白盒测试”的必要工作，程序员应该在前期的程序设计中，就埋下很多性能测试的功能模块，使程序运行的每一个环节，信息尽量可

见，在产品进入正式测试之前，就解决绝大多数性能问题和 bug。这里面的技巧包括如下内容：

1、养成随时统计的习惯，在笔者做传输模块时，很多时候习惯性的都要设置一些线程安全的变量，脱离业务之外，统计传输的字节数，报文数等数据，这些基础数据，一旦经过与时间的计算，都可以迅速显示当前的包交换率、丢包率、传输带宽、吞吐量等基础数据。

2、数据保持可见性，笔者设计的传输模块类，一般习惯性的都有一个 `PrintfInfo`，或者 `PrintfInside` 的函数，该函数在正式工程中一般不起作用，但在测试阶段，笔者习惯开辟一个线程，以 2 秒左右的频率定时输出，保持内部数据可见性，效果非常良好。

3、必要的统计工具准备，笔者在设计基本库时，将一些常用的统计模块，纳入 debug 基本库模块，随时调用，这些工具和习惯，均很好地保证了工程代码的稳健性和问题的早期发现和解决，在后文中将有详细介绍。

根据笔者的经验，白盒测试实现性能跟踪的关键，就是中间输出，将一些关键的参数，在程序运行的中间以某种方式循环输出，可以简单打印到屏幕上，也可以输出到文件中。

在这种循环输出的模式下，很多的性能问题，仅仅靠程序员肉眼观察，已经能够发现并基本定位，这可以使程序员及时处理潜在的 bug，避免后期测试时出现大问题。

举个例子：这里给大家提示一段笔者近期完成的一个服务器工程的中间输出信息，给大家建立一个感性认识。

```
*****
listen pool: accept task count=0           //listen 池正在响应的连接请求
up pool: task count=0                       //上行链路任务数
down pool: token count=0                   //下行链路任务数
task pool: thread count=15, task in queue=3 //任务池线程数和任务队列淤塞情况
task run: task count=4                     //正在执行的任务统计
socket pool: 2 / 3, biggest=4              //socket 注册池在册的 socket 数，
                                           //以及最大 socket 编号
memory pool: 45 / 48, biggest=0xb6649010   //内存池信息
block=32, use=1026 kbytes, biggest=0x99834c0 //块数，内存使用量，最大指针
[16] stack: all=1, out=0, in=1             //16 字节的内存块数量
[32] stack: all=17, out=15, in=2            //32 字节的内存块数量
[128] stack: all=9, out=9, in=0             //128 字节的内存块数量
[256] stack: all=4, out=4, in=0             //256 字节的内存块数量
[1048576] stack: all=1, out=1, in=0         //1024k 字节的内存块数量
*****
```

以上信息，在服务器运行时，每 2 秒打印一次，随时显示运行情况。简单说，其中的 task pool，就是任务池中的任务，socket pool，是目前使用的 socket 的统计，memory pool，则是内存池的使用状况。这些信息在本书中并无实际意义，仅提供给各位读者一个感性认识，其中内容具体表示什么含义，并不重要。

这里请读者关注的是观察的方法。在实际观察时，笔者往往并不看重某些数值的绝对值是多少，而是关心这些数字是不是稳定，有没有恶性增长，即只增不减，这是最关键的。

我们知道，服务器要求 7*24 小时运行，很多时候，内存、socket、线程等资源哪怕不泄露，但是在完成服务后没有及时释放，占用的资源越来越多，最后也会导致服务器挂死，无法服务的 bug。这也是商用服务器的开发比普通程序开发要求严格得多的重要原因。

因此，这类循环输出的真实目的，是帮助程序员即使观察到某一个单项指标是否有恶性增量，在某个峰值之后，其占用的资源能否被及时释放，这些都是保证服务器，或者嵌入式设备等 7*24 小时运行设备，能长期工作的关键。

当然，我们知道，当这种直观的观察结果，所有指标一切正常，申请和释放逻辑工作无误，我们当然也可以推论，程序没有任何潜在的资源泄漏，在这方面没有 bug。

2.7.3 随机数的产生

在讨论统计信息是，讨论随机数，可能读者会觉得有点文不对题。毕竟是风马牛不相及的事情。不过，根据笔者的经验，发现很多时候，性能统计、debug 和随机数的使用，几乎可以说密不可分。

这也很好理解，性能统计，有两个必要条件才有意义，一个是我们有统计的手段，另外一个是有足够模拟实际工作的任务压力，这包括带宽压力，业务压力，风暴压力等等。而这些压力的模拟，其实必须用到随机数。

这里笔者仅仅提醒大家一点，在工程库中，随机数的产生和使用函数，基本上都归类到性能测试这个模块中。

2.8 队列无处不在

相信各位读者在学校和教科书中，学习了大量的数据结构和算法。不过，在商用并行工程这么多年的实战中，笔者发现，队列是用的最多的，几乎可以说无处不在。因此，对队列的熟练使用，可以说是商用工程的重中之重。

2.8.1 数据结构在数据传输中的应用分析

这也很好理解，队列的先进先出（FIFO）特性，确保了信令顺序不混乱，另外，队列相当于一个缓冲区，可以用来暂存数据，因此，很多情况下，必须使用队列：

1、异步任务的传递，需要队列。一个线程，出于某种目的，希望把手边的任务交给另外一个线程处理，一般都是使用队列做缓冲。

2、高低速计算的缓冲，需要队列。CPU、内存和总线，属于高速设备，而网卡、磁盘这些外设，属于低速设备，当高速设备完成计算，向低速设备发送，则需要使用队列缓冲。

3、信令保护，需要队列。很多时候，网络中间服务设备，如中继服务器、路由器等，为了确保每条报文信令都不丢失，需要实现存储转发逻辑，以及等停式工作模型，需要用队列做缓存。

4、数据打包工作，需要队列。传输中，为了尽量提升传输效率，很多时候，需要将多条信令打包为一个报文，进行高速收发，而队列的先天特性，对于这类保证顺序的打包解包帮助很多，很多时候需要用队列完成。

5、并行计算内部的任务机，需要队列。商用数据传输工程，很多时候涉及并行计算，原则上都是“任务池+任务机”工作模型，其核心一般都是一个任务队列，供中心并行计算元与周边计算模块实施任务交互。

2.8.2 我们需要哪几种队列形式

队列有很多种实现方式，笔者根据经验，总结出至少以下几种队列都是需要的，这在后文都有详细的代码说明。

1、动态队列：动态队列依托内存池，使用动态内存分配，内部实施优化，实现高速的入队和出队控制。这几乎是最常用的一种队列形式，几乎每个缓冲性队列都需要使用动态队列实现。

2、静态队列，静态队列依托一个静态内存区，内部以队列方式管理，最终队列数据是一个完成的二进制缓冲区，这在数据打包中非常必要，可以大大减少打包的开发压力。

3、文件型队列，队列作为网络信令的中间缓存模块，很多时候会由于某些网络故障，导致大量的信令淤积，仅仅使用内存队列的话，很容易溢出，导致信令丢失，因此，商用数据传输工程，对于某些需要高保证级别的信令流，一般都使用文件型队列来实现缓冲，尽量减少溢出的可能性，保证系统安全运行。

2.9 不要求全责备

在本章的最后，还是要和各位读者沟通一些细节。至少笔者认为，在商用工程项目开发中，这是非常关键的一个问题。

我们前面已经说了，商用分布式并行系统，应对的是互联网上各种复杂的网络环境，这说明两个问题，一是实验室测试的结果，并不代表实际运行的结果，二者没有可比性，但另一方面也说明，工程中永远没有完美，永远没有 100%正确的系统。

在笔者很多次项目实战开发中，往往到最后快要结束的时候，在系统的性能、稳定性的某一方面，达到一个瓶颈状态，即不管如何优化，性能不升反降。给笔者的感觉，好像一个二次曲线，已经达到顶峰，任何调整，都向更坏的方向发展。

笔者认为这是正常现象，商用分布式系统的数据报文，是直接在互联网上流动，途径不同的节点路由器，各种防火墙、网关等，每个节点，每时每刻的任务负载都不一样，因此，很难有个标准的环境来测试一个标准的结果，很多时候，测试的结果并不是 0 和 1 的简单区别，而是在 0 和 1 之间，有大量的灰色地带，是一个百分比的问题。

这在很多具有学生思维的读者看来，可能有点难以理解，在学校里面，很多时候要求我们一定要做足 100 分，一个知识点，会就是会，不会就不会，没有什么中间概念。这个思想，在学校中，是正确的，但在企业中，就不一定准确了。

一个商用系统，最终的性能指标，一般都是以百分比表示的一个浮点数字，如一款路由器产品，可以有丢包率，可以有故障率，这些都是合理的，在实际的销售过程中，客户也都能接受，因为谁也不敢保证，自己的网络状况 100%正常。因此，建议各位读者在以后的工作中，要学习正确对待系统的不完美性，不要求全责备。

很多程序员，在商用工程开发结束，进入测试期间，往往会陷入一种莫名的恐慌，因为 QA 报出来的 bug 很多时候，无法分析，找不到原因。在笔者看来，其实这是正常现象，QA 很可能把某一次网络阻塞导致的异常现象，当做 bug 报出来。

举个例子：笔者在开发 http tunnel 防火墙隧道穿越项目，在测试时遇到一个事件，本来测试已经接近尾声，但某一天 QA 突然报 bug，丢包率大增，语音通信断断续续，系统性能变得很坏，笔者一看，确实如此，只有重新开始 debug。

但很奇怪，无论笔者怎么跟踪，调试，最后效果都是越来越坏，刚开始还可以 6~7 路并发，但后来连两路都无法维持，但到下午下班前一刻钟，所有问题解除，一切又正常了。

笔者感到很奇怪，开始怀疑是公司内网问题，就第二天请网管检查，结果发现有 6 个人在使用 BT 下载，那几天整个内网带宽几乎被占完，路由器的出口带宽也基本上被抢光，那程序肯定无法正常运行，后来经过网管警告，大家不再使用 BT 类下载工具，故障自动解除。

大家可能不信，就这么一件事情，笔者花了两天时间。

不过，有了这次经验，笔者以后处理类似情况就简单多了，还是这个工程，在一年后的一次中期修订时，QA 又爆出来类似错误，笔者观察到报错误的时间是上午 9:00 左右，刚好是公司上班的时间，于是就让 QA 再观察半个小时再报。

半小时后，QA 兴奋地报告，一切故障解除，究其原因，9:00 是上班高峰期，是大家开机时间，很多网络应用软件如 QQ、MSN 等，开机时会进行注册登陆，自动检测版本更新，下载新版本等工作，这期间网络较为繁忙，过了这个时间段，自然就好了。

大家通过这个事例，是不是可以联想到什么？

作为商用程序员，做事情力求完美，是正确的，但笔者在这里提醒一句，当系统设计经过评审讨论，逻辑完全正确，当我们使用后文的无错化程序设计思想，完美地实现了我们的代码，当我们已经获得了较好的实验结果，那么，我们应该对自己的程序保持一种天生的信心。

笔者认为，作为商用数据传输工程的程序员，甚至所有商用工程的程序员，最重要的应该具有冷静的头脑，特别是在遭遇某些异常情况的时候，能静下心来，客观仔细地分析问题，把所有的可能原因都逐一分析，是自己的问题，无条件修改，不是自己的问题，也要勇于陈述，取得团队成员的共识，最终，保证项目如期完成。

第3章 C/C++语言无错化程序设计

- 3.1 “无错化程序设计”简介
 - 3.1.1 无错化程序设计思路
 - 3.1.2 C/C++无错化设计的解决方案
 - 3.1.3 使用后的效果介绍
- 3.2 计算机程序的真谛
 - 3.2.1 程序就是“搬数”
 - 3.2.2 程序就是“写文章”
 - 3.2.3 程序就是“复制”
 - 3.2.4 笔者看程序设计
- 3.3 定名
 - 3.3.1 匈牙利命名法
 - 3.3.2 函数命名原则
 - 3.3.3 变量命名原则
 - 3.3.4 其他命名规则
 - 3.3.5 定名的折中
- 3.4 无错化程序的基本书写原则
 - 3.4.1 写简单易懂的程序
 - 3.4.2 严禁变量转义
 - 3.4.3 严禁一语多义
 - 3.4.4 函数只能有一个出口
 - 3.4.5 变量如不使用，保持初值
 - 3.4.5 常量必须定名
 - 3.4.6 太大数组不要用静态方式
 - 3.4.7 尽量避免使用递归
 - 3.4.8 解决方案一套就够
- 3.5 基本程序设计原则
 - 3.5.1 函数的设计
 - 3.5.2 类的设计
 - 3.5.3 其他要点
- 3.6 基本语句的约定
 - 3.6.1 判断语句，常量永远在左边
 - 3.6.2 `for(i=0;i<n;i++)`
 - 3.6.3 `while(1)`
 - 3.6.4 不要使用 `do...while()`
 - 3.6.5 `i++`和`++i`问题
 - 3.6.6 请不要使用“`?(...):(...)`”结构
 - 3.6.7 善用大括号`{ }`缩小作用域
- 3.7 请使用 `goto` 语句
 - 3.7.1 函数只有一个出口的原则需要 `goto`
 - 3.7.2 谁分配、谁释放的原则需要 `goto`
 - 3.7.3 商用工程要求 `goto`
 - 3.7.4 程序的易读性要求 `goto`
 - 3.7.5 `break` 为什么不能乱用

- 3.7.6 goto 的常规使用手法
- 3.8 指针的使用原则
 - 3.8.1 商用数据传输常见的指针类型
 - 3.8.2 不要使用两个以上的*号
 - 3.8.3 指针不能参与四则运算
- 3.9 使用结构体的技巧
 - 3.9.1 结构体传参的必要性
 - 3.9.2 预防多重指针的隐患
 - 3.9.3 32 位到 64 位移植
 - 3.9.4 弹性内存使用需要结构体传参
 - 3.9.5 网络传输协议，需要结构体传参
- 3.10 使用宏的建议
 - 3.10.1 宏的几大作用
 - 3.10.2 C++的建议
 - 3.10.3 编译宏—跨平台开发
- 3.11 回调函数设计方法
 - 3.11.1 回调模型设计者
 - 3.11.2 回调模型使用者
 - 3.11.3 参数传递的常规手法
 - 3.11.3 事件模型和回调模型
- 3.12 C 语言字符串的深入研究
 - 3.12.1 字符串拷贝
 - 3.12.2 字符串构造
 - 3.12.3 关于字符串处理的结论
- 3.13 C/C++语言无错化程序设计小结

第 3 章 C/C++语言无错化程序设计

商用工程，一般都是相对比较“大”的工程项目，一个项目，需要集合几个人、十几个人、甚至几百人合作完成。另一方面，一个程序员，一旦参与到商用数据传输工程中，其接到的需求、涉及到的算法复杂度、数据规模和技术深度，都远大于一个传统的单机项目。

这说明，一个商用程序员，一般都会碰到较大和较多的需求任务，这些任务涉及大量的业务逻辑的实现，针对业务上的开发会占到商用程序员很多的开发时间。

举个例子，一个网络游戏的程序员，其大量的时间是花在游戏效果、耐玩度、特效等游戏业务相关的程序实现上，而一个 ERP 程序员，则可能大量地需要了解用户的业务流程，工作流走向，相应的规章制度等。

这说明，一个商用数据传输工程的程序员，可能在开发中很少有时间去精雕细琢自己的代码，很难像在学校一样，花大量的时间和精力，来关注程序本身的 bug。这导致商用代码程序员，在刚开始的时候，会感觉到很累。

很多年以前，当笔者进入 IT 行业之前，就听说这个行业非常苦，学不完的新技术，看不完的书，还有，就是无穷无尽加班。在工作了四五年之后，笔者发现这个说法确实是有道理的，真的很累，不止是身体累，而且心累。

当时笔者就在想，有没有一种方法，能让笔者这类程序员不要太累，可以过的轻松一点。为此，笔者经过了长达数年的思考和研究。

经过笔者研究，发现自己很多加班都来自于 bug，而其中最大的时间开销，并不是解决 bug 本身，而是查找 bug，

因此，笔者就试图研究一种方法，在开发期间能有效降低 bug 的数量，并且尽量使 bug 提早自我暴露，便于查找，经过试用，效果很好，在之后的工作中，大幅度减少了工程中的 bug。

这种方法，笔者称之为“C/C++无错化程序设计方法”。

这种《C/C++无错化程序设计方法》，笔者曾经在 2003 年，在西南交通大学计算机学院做过演讲，当时获得了同学们的一致好评，经过进一步总结，笔者把它视为商用数据传输工程开发，实现无错化代码的必要条件，在本书下面的章节，将做重点介绍。

提示：无错化程序设计的核心思想，并不是绝对没有 bug，而是首先承认，程序员是人，是人就会有 bug，在这个共识下，我们来讨论通过一些方法，使 bug 更加易于暴露，尽早暴露，降低 bug 解决的成本。

3.1 “无错化程序设计”简介

在笔者刚刚进入研发领域的头几年，确实如前所述，bug 很多，工作感到很累，不过由于笔者学了一点管理学的技巧，学会了用统计分析的方法解决问题，于是有一天突发奇想，想统计一下，造成笔者这么累的真实原因有哪些？

经过对过去商用工程项目的统计，笔者发现主要原因如下：

- 1、新技术层出不穷，学起来疲于奔命。
- 2、自己的产品 bug 太多，80%左右的加班都来自于查找 bug。
- 3、部分原因是公司任务紧，但这种情况不到 10%。
- 4、偶尔会有系统框架设计错误，导致整个程序重构，但这种情况更罕见，不到 1%。

根据这些原因，笔者做了进一步思考，提出了如下解决方案

3.1.1 无错化程序设计思路

根据管理的原则，不怕有问题，出了问题，做针对性解决就好了，因此，笔者做了进一步分析，看能不能提出解决方案：

1、笔者经历过的几乎每个公司都有任务紧的情况，只有一个公司例外，但那家公司因为经营不善已经倒闭了。因此，任务紧导致加班这种情况是行业惯例，笔者只能认命。但好在这些情况其实并不多。

2、新技术其实很多都是老技术的翻版，很多新技术仅仅是利用老技术，将新的应用做了一层抽象，如果熟练掌握老技术的话，自己推导有时候不是很难。比如，COM 技术的本质，还是实现跨进程的通讯，进而实现自动传参，实现远程过程调用。这些在传统的 socket 通信中都有解决方案。

3、很多时候，学习新技术的动力其实仅仅来自于招聘面试官的一句问话，或者某个提法，但进入公司之后，发现并不是绝对因素，公司内部大量使用的还是成熟代码。可以考虑先了解概念，待具体使用时再学习原理。

4、bug 多这个问题应该认真对待了，这是加班的主要因素，如果笔者能做出 0 bug 的程序，那么，80%的加班将不存在。

5、笔者自己最大的 bug，就是 Windows 的蓝屏，内存泄漏，socket 泄露这些问题，还有就是线程锁问题，如果把这些问题解决掉，则笔者有信心解决掉 80%~90%的 bug。

6、去除 bug 最大的问题不是改代码，而是查找 bug，如果能考虑实现一套机制，自动将 bug 暴露出来，则 bug 将不再是 bug，可以在代码阶段全部解决。

7、很多 bug 都是小 bug，由于没有及时在单元测试中检查出来，带到系统集成中，产生大问题，而且很不好找，因此，模块级的白盒测试至关重要。

8、系统分析和设计问题，属于笔者个人经验问题，只有不断多做工作，增加自己的积累，来慢慢改善。暂时不做处理。

进一步思考，笔者发现不同的代码的难度不等，代码普遍分为“实现”和“调用”两类代码，毋庸置疑，“实现代码”的难度较高，因为为了适应各种调用，抽象度高，书写困难。笔者曾形象地称之为“艰难代码”，与之对应的调用代码，属于“容易代码”，即比较好写，也不太容易出 bug。

笔者思考，如果将自己已经写过的“实现代码”进一步抽象，做成工程库会怎么样？显然，当时开发可能会更累一点，因为要把一次性代码改写成抽象度更高的公用性代码，会很难，但只要坚持，则几年以后，笔者可以拥有一个适应自己主要研发方向的基本工程库，那么，再开发新的应用，就不再是难事。

而且，工程库代码都是经过了数个商用工程验证过的，稳定性自然高，新工程的 bug 也能控制住。

3.1.2 C/C++无错化设计的解决方案

经过思考，笔者整理了如下的解决方案，来帮助自己实现无错化的程序设计：

1、接到任务，先做需求分析，理清所有需求和基本解决方案，形成文档，和客户充分沟通确认。在需求分析阶段，确定各个关键数据的边界，进而确定内部各个关键边界值，进而根据数据规模基本确定算法。

2、严格按照系统分析步骤走，先画拓扑图，再画时序图，数据流程描述等必须完善。模块设计至少做三层，关键算法一定有流程图保证，核心数据确保在设计阶段确认完毕，根据项目特点，和客户、主管、同事充分讨论沟通，确保无逻辑硬伤。

3、养成良好的开发习惯，包括抽象思维，每个逻辑只写一遍，尽量通用，良好的命名规范，良好的编程习惯，固定一种有效的程序设计风格，不要变来变去。每个函数、模块做好防御性设计，对输入参数严格校验，不符合规范即拒绝服务。

4、解决方案一套就够，不要贪大，贪新，尽量用成熟算法和技术解决问题，如非必要，回避对新技术的使用，避免引入不必要的 bug。

5、做事情坚持“先解决有无问题，再解决好坏问题”的原则，永远是先实现，后优化。

6、建立自己的工程开发库，代码成熟一段，入库一段，成熟的标准为商用工程交付客户一年以上无 bug。严格实施基于工程库开发方式，入库代码原则上不许修改，只准扩充。库中已有算法，严禁再次开发，避免出现重复的 bug。库代码要求高度抽象，具有跨平台、甚至跨语言的可重用性。

7、建立程序自纠错机制，所有“艰难代码”自带 Debug 功能，可以简单实现 bug 的自动报警和输出。重点是各种资源（内存、socket、锁等资源）的监视和控制。建立完善的 Debug 和 log 日志系统，系统状态随时输出调试。

8、充分重视单元模块白盒测试，测试代码与工程代码比例达到 1: 1 左右。每一段代码，三轮单元测试无法调试通过，即废弃重构，避免因惯性思维导致顽固 bug，每次工程完工，测试代码和因测试抛弃的废代码达到总工程量 50%~60%左右。

9、整合测试以数量保证，商用数据传输工程，整合后至少保证 500 万~1 亿以上的报文风暴冲击，确保整体系统的合理性和正确性。

10、开发中，坚持“**以一个好的习惯，解决整整一个方面的 bug**”的原则，对于某些确实无法以程序开发方式解决的 bug，强迫自己养成良好的习惯来解决。如对于锁的正确使用等。

11、善用版本管理，自己给自己搭建版本管理环境，如 svn 服务器等，方便库代码和工程代码的维护。

在本章中，笔者将着重针对如何书写正确的代码，给大家提出一些原则。并在后文的示例中一一介绍。

3.1.3 使用后的效果介绍

笔者大约在 2000 年左右，开始使用上述方法，逐渐建立自己的基本库，构建自己的基本程序书写方式和开发习惯。

前期的工作确实很累，笔者在完成正常的工作的同时，还花了大量的业余时间来完善自己的基本工程库，这个库，笔者把它看做是自己写给自己的产品，因此写起来精益求精，仅仅一个多线程安全的变量模板，笔者写了 18 遍，一个应答 Socket 传输模块，笔者写了差不多 38 遍，至今都还在持续完善中。

但这个方法使用之后，效果显著，举几个例子：

1、2002 年，笔者进入四川迈普公司的时候，接到的第一个任务的开发一套报表模块，《ReportMaster》，一个大型网管数据统计系统，统计的范围差不多 20 亿条，笔者一个月左右完成，提交 QA 部门后，仅仅查出 1 个 bug。获得公司和同事的好评。

2、笔者近期带领团队，开发的一套服务器集群系统，十几万行代码，经过 alpha 到 beta 共计 9 轮测试，QA 部门报出的 bug 为 51 个，其中，属于 C/C++ 部分的 bug 仅有 7 个。也获得了公司的肯定。

3.2 计算机程序的真谛

之所以写这一节，是因为笔者认为，很多无错化程序设计的思维，其实是来自于笔者对于程序开发这门科学最原始的思考，此处需要论述一下。

从 1986 年，首次接触 Apple II 计算机开始，笔者从事计算机软件开发有 20 年左右的历史，从最早的 Basic、6502 汇编语言，到现在的 C、C++，笔者一直在思考一个问题，“程序设计究竟是一个什么样的工作？”，这个问题困扰了笔者很多年，期间也看过很多专门的论述，但笔者经过思考，有了自己的一点答案。

3.2.1 程序就是“搬数”

笔者认为，程序，首先就是“搬数字”。把一堆数字，从一个地方，按照一定顺序，读出来，然后，加上一个计算动作，放到另外一个地方，仅此而已，所有的计算程序，莫过于此。

举个例子，一个游戏程序，通常其程序开发的核心是视频、音频、操控性（这里暂时不谈企划、故事情节，因为这些可以算作程序以外的东东）。

视频，很简单，内部保留的原始贴图，3D 模型，在运行时被动态读出，经过一定运算，最终把数字放到显卡的显存中，实现显示。

音频则是读出数据，按时序打入声卡的接口，最终播放出来。

操控性，则是从输入设备（键盘、鼠标等等），读取数字，然后经过计算，最终转化为图形和声音的计算参量（如转弯、执行、发射等），参与到图形和声音的计算中，最终帮助图形和声音子系统，计算出正确的结果，输出到合适的设备。

再举个例子，一个 IM 系统，如 QQ，一般是从键盘或鼠标等输入设备读入数字（文字也是数字的一种），然后经过一定的编码计算，发送到远方另外一个客户端，再经过逆向解码，显示到对方屏幕上。

所以，笔者认为，无论什么平台，无论什么操作系统，无论什么语言，计算机程序唯一做的事情，就是“搬数”。每段程序都在做“读取—计算—写出”这三个动作。当然，这个数据读取过程很可能是传参进入。

正是因为这个原因，笔者认为，算法语言和数据结构，异常重要，数据结构决定了数据如何组织，搬动起来效率最高，而算法语言决定了如何搬效率最高的问题。在很多次年轻的同学的请教过程中，笔者都建议对方先不忙学习语言，先把这两门课学习好。

既然程序的核心任何是“搬数”，那我们就可以从直觉上，认为程序设计中，有如下要点必须要重点关注，才能写出无错化的程序：

- 1、数据的边界，任何一个系统，资源都不是无穷无尽的，因此，不可能处理无边界的数，如果我们在设计一个商用系统时，没有边界意识，对程序行为没有边界控制，那么，程序写出来就是一场灾难。因此，系统分析阶段，落实每个关键数据的边界，是至关重要的一件事，并且要确保每个边界被落实到程序设计中。

- 2、进入数据的安全性，任何一段程序的中间计算，将依赖于读入过程进入的数据，如果进入的数据不正确，不仅仅后续结果无法保证，连程序本身运行是否安全都是个问号。因此任何一段程序，必须严格校验进入的数据，一旦异常，立即终止服务，以降低服务品质来保证系统的安全性。商用程序，更像一个过滤器，无论外界给出的数据正确与否，最终都能保证自身的安全，仅仅是错误的数字不提供计算服务而已。

- 3、数据的精度，我们看到的任何一个计算机系统，其实都是整型计算机系统，内存单元一定是整数表示，因此，所有的浮点数，都是通过算法模拟出来的，绝对的精度是不存在的，另一方面，我们大多数商用计算，都有预设的精度需求，超精度计算没有任何必要，因此，确定整个系统的计算精度需求，并落实到开发实战中，是系统设计的重要任务。

- 4、安全地输出，我们不能保证给我们的数据，都是正确的，但作为商用代码，至少应该确保自己输出的数字，一定是正确的，合理的。这包括每次调用其他子程序的参数，必须正确，函数的返回值，一定真实体现设计需求，在合适的状态返回合适的值。

根据笔者的经验，我们只要做到上述几点，其实程序计算逻辑上，人为 bug 已经很少了。这要求我们系统分析阶段，严格界定每个关键数据的边界，包括空间边界和时间边界，并确定精度，每个函数、每个类模块，对于输入的参数做严格校验，对于自己提供给别的模块的参数，要通过仔细斟酌，尽力保证正确性。

3.2.2 程序就是“写文章”

做到以上几点，单就写一个简单的实验程序来说，其实基本上已经够了，程序一般没有大问题。但就商用工程来说，仅仅靠以上几点，还远远不能写出符合商用代码规范的程序。

原因很简单，商用工程，一般都是多人合作的项目，程序员直接的交叉开发司空见惯，商用代码，必须满足“双可读性”，即不仅仅计算机读起来没有错误，还要做到人读起来无障碍。

在笔者多年的职业生涯中，遇到过太多运行无错误，但是人读起来艰难无比的代码，给工作带来无数困扰，很多时候，不得不因此而重新开发。这在某些开源代码中，尤其明显。

很多业余开源代码（像 SUN 公司主动开源的产品不算），通常是由开源社区维护的，参与开发的每个程序员，一般都没有从中获利，大家完全凭兴趣来完成工作。

这就带来一个问题，开源代码的开发者，没有任何生存压力和责任感，代码质量低下，同时，某些开源程序员，可能出于某种技术保密的策略，对于某些关键算法，在保证程序逻辑正确的情况下，故意做某种模糊处理，使程序的可读性变差，如使用缩略，无意义的命名等。

而另外一方面，商用代码程序员，在公司内一般都有严格的任务压力，并且有代码规范性要求，写出代码的可读性和正确性，一般和自己的薪水、职位挂钩，并且后期还有严格的 QA 测试保证代码质量，因此，商用代码的质量，一般远高于开源代码。

举个例子，笔者在阅读著名的 VoIP 软交换机 Asterisk 的源代码时，很多时候被一些莫名其妙的变量命名弄混，并且没有注释，非常头疼，更有甚者，开发者出于某种目的，故意使用纯 C 来实现了类似于 C++ 的面向对象开发模型，其中大量的回调函数注册机制使程序几乎不可读，不可跟踪，并且还美其名曰这是“Unix 程序员的传统”。这严重制约了后人对这个软件的掌握程度，其实反过来，也制约了这个软件的商业化推广。

这也是笔者在商用开发中，一直小心谨慎使用开源代码的原因。笔者认为，开源代码不是不能用，但一定要谨慎，选择时要看来源，看社区的口碑度，看文档和代码的详细和规范程序，并尽量自行做好相关的品质和性能测试，方能安心使用。

在笔者看来，程序其实很简单，完全没必要写得这么晦涩难懂。写程序，就是做文章。写一段话，就是让人看了，知道先做什么，再做什么，最后得到什么，得到的结果有什么意义。

写一段程序，就是让计算机看了，知道先做什么，再做什么，最后得到什么，得到的结果有什么意义。

请大家看看，这两句话的含义有差别吗？这说明，如果我们写出了人容易看懂的程序，其实计算机也好懂，人一眼看起来没有错误的程序，其实计算机看起来，也没什么错误。

笔者在无错化程序设计方法中，特别强调命名的严格性，就是希望通过命名，来使程序的逻辑从人类的语法上，讲得通，看起来一目了然，避免程序晦涩难懂造成的 bug。

举个例子：

这是一段字符串拷贝的函数，目的是自动检测来源和目的指针的长度，避免读出界和写出界，并且自动在拷贝的目标数据最后，根据 C 语言的字符串规则，尾部添加 '\0'。两段函数的逻辑完全一样，仅仅是命名差异，大家可以比较一下读出来的效果。

第一段：

```
void scp(char *p1, char *p2,int n)      //函数名缩写，晦涩不清
{
    int n1=(int)strlen(p2)+1;           //n1,n,p1,p2 具体什么应用，没有说明
    if(!p1) goto scp_1;                 //标签定义随心所欲
    if(!p2) goto scp_1;
    if(n1>n) n1=n;                       //看不懂这个计算的含义
    memcpy(p1,p2,n1);                   //从这句话猜测，p1 是目的地址，p2 是源
    *(p1+n1-1)='\0';                   //这句话的计算无法读懂
scp_1:
    return;
}
```

第二段：

```

//安全的字符串拷贝函数
//参数按顺序排：目的地址，源地址，拷贝的最大字节数
void SafeStrcpy(char *pDest, char *pSource,int nCount)
{
    int nLen=(int)strlen(pSource)+1;           //获得源字符串长度，+1 保证'\0'
    if(!pDest) goto SafeStrcpy_END_PROCESS;    //保护措施，如果目标地址非法
    if(!pSource) goto SafeStrcpy_END_PROCESS;  //则放弃服务，跳到结束点
    if(nLen>nCount) nLen=nCount;               //避免读写出界的设计
    memcpy(pDest,pSource,nLen);                //实施拷贝
    *(pDest+nLen-1)='\0';                      //手工添加'\0'
SafeStrcpy_END_PROCESS:                       //这是结束点
    return;
}

```

大家可以看到，同样的逻辑，仅仅是换用了有意义的命名，整个程序的可读性一下就增加了，程序看起来一目了然，不容易出 bug。

不过很不幸，在大量的开源代码中，大家阅读到的，很可能是前一种命名习惯。显然对阅读造成很大困扰。

请大家以后写程序时注意，显式的定名程序中每一个接触到的变量、函数、宏、类，以有意义的名字替代无意义的代称和缩写，定名中尽量携带尽可能多的信息，帮助后续的开发者的理解，把程序写得像一本小说一样，或者像一本工艺说明书一样，这样的程序，会非常优秀。所以，笔者通常说，程序设计，从“定名”开始，高质量的程序设计，从“易读”开始。

3.2.3 程序就是“复制”

笔者有次统计了一下开发这么多年，涉及的算法和数据结构，很有趣的是，笔者发现“队列”是用得最多的。而在程序设计的基本逻辑中，循环和判断又是用的最多的。这可能和笔者做到的具体工程项目有关。

不过，推而广之，笔者认为，每个程序员，从事一个行业，只要做得多了，一般都能总结出自己行业最常用的算法和数据结构，自己日常的工作，可能就是不断“复制”这些常用的算法和数据结构。

这说明，如果我们能建立自己的常用工程程序库，将常用算法和程序经过多年总结和归纳，使之具有高可用性和高度正确性，则我们可以大大缓解后期我们的工作压力。而现代程序开发，各种语言也是鼓励大家这么做，如 C++ 的泛型程序设计，其实主要就是针对工程库开发而设计的。

事实上，笔者就是这么做的，在本书中，笔者将向大家 share 一个笔者多年总结出来的传输库，大家可以学习一下如何建立自己的工程库。

在一个商用工程项目团队已经拥有一个工程库的基础上，则要求这个团队以某种规章制度将库的使用固化下来，一般原则如下：

- 1、每段代码仅允许书写一次，如果没有特别的理由，如库代码无法满足新工程需求，则不允许对成熟代码的进行二次书写。

- 2、安排专人负责工程库的收集、维护和发放。这也通常是项目团队的核心骨干。

- 3、每次工程完毕，总结阶段，要收集本次工程可以入库的新代码，通过团队讨论，进行抽象性维护后入库。

3.2.4 笔者看程序设计

综上所述，笔者认为，如果要书写无错化的代码，必须遵循以下的步骤：

- 1、详细的系统分析设计，落实每个模块的数据处理边界，算法精度，模块间的每条互动关系，数据流向，每笔特征业务数据的边界。
- 2、详尽的定名法则，以某种团队公认的命名法，为程序设计中处理的每个元素定出有意义的名字。
- 3、关键算法的清晰描述，原则上，在系统设计任务书上，应该对关键算法做出清晰的定义，包括流程说明，说明时关注对数据边界和异常处理原则的说明。
- 4、严格的书写规范，以团队内部成员看得懂的形式，显式定义一种书写规范，确保大家用相同的方言说话，确保沟通无障碍。
- 5、在此基础上，安排专人负责工程库的收集和维护，以及工程库代码的发放工作，并持之以恒地坚持下去，最终实现整个商用工程项目团队的无错化设计。
- 6、以上措施，程序员也可以自行完成，不过建议参与到团队建设中。

当然，本书定位是一本供程序员开发的参考性书籍，因此在后续章节中，笔者更多地会就程序设计方法学，和各位读者展开讨论。而不会对项目开发团队的建设 and 系统分析做过多讨论，各位读者有兴趣的话，可以自行参考相关专业书籍进行学习实践。

3.3 定名

在前文我们提过，“**程序设计，从定名开始**”，这个“定名”，包括对所有变量、常量、函数、类、宏等程序开发涉及到的元素，定出一个精准的名称，以保证后续程序书写的直观性和可读性。

但我们知道，通常一个商用工程项目，很难有太多的时间做详细的系统分析和设计，一般都是写个大概，然后一边写程序，一边修订系统设计书，至少笔者经历过的所有公司，都是如此。这说明，“定名”这个工作，其实是贯穿于商用工程项目开发的始终，很难在初期设计阶段，即将所有元素定名完毕。

因此，笔者强调在一个商用工程项目的实施中，应该确立一个定名原则，所有的项目程序员均按此原则进行定名，并由专人负责检查定名效果，这比仅仅在系统设计一个阶段实现定名要实用得多。

定名还有个重要作用，就是作为注释的补充。笔者在很多公司都看见了对于注释的严格规定，但无可否认，这些规定其实执行得都不好，原因很简单，在忙碌的商用工程开发中，程序员 90% 的精力都在关注业务、bug，很难有精力再来关注程序注释是否足够，后来的阅读者是否看懂的问题。包括笔者自己也是这样。

而不开玩笑地说，几乎每个商用工程项目的总结会都是聚餐之类的集体休闲活动，也很少有公司或团队，能认真对待项目总结，Review 甚至 ReCoding 代码，使之更加易读。

这说明，要求程序员来完成详尽的注释，本身就是一件不太可能的任务，有点太过于理想化，大多数情况下，不具备可操作性。因此，笔者在带领团队开发的过程中，强调通过定名，使程序尽量“自解释”。一段程序，哪怕没有足够的注释，至少通过定名，也能大概看懂其含义。

3.3.1 匈牙利命名法

笔者由于是学习 VC++ 进入 C++ 领域的，因此对微软公司推荐的匈牙利命名法比较习惯，在日常工作中，也较多采用这种命名法定名。由于笔者无意抄袭前人的文章，因此匈

牙利命名法的具体规则，笔者在本书中就不一一细述，有兴趣的读者，可以自行在网络上查询，或参考相关文档、书籍。

但有个有趣的现象值得一提，笔者在网上和一些朋友讨论时，特别是在 CSDN 的学生群中，收到很多反对意见，这让笔者非常意外。

经过总结和分析，笔者发现大多数反对者，主要针对的是匈牙利命名法不符合英文书写习惯，以及并不是其他语言推荐的输入法，不太符合很多开发者的书写习惯。很多网友也举出了很多匈牙利命名法不严谨之处，这些笔者也同意。

不过，针对商用工程开发的“定名”这个问题，笔者是这么理解的，命名法其实就是一种项目团队内部的公共语言，好比汉语中的普通话，其选择标准并不要求非常严谨和科学，但一定要有普适性，因为其中心思想是为实现团队内部程序模块共享，以及降低内部沟通成本来服务的。换言之，笔者认为，命名法是为需求服务的。

匈牙利命名法仅仅是众多命名法的一种，笔者也并不强调所有读者必须遵守，大家完全可以根据自己所在的项目团队的习惯，以及历史积累的写法，自行选择命名法，只要团队内部成员大多数都认可就够了。

本书由于使用了大量笔者过去的代码，因此不可避免地使用了匈牙利命名法，读者完全不必认为这是必须的。

事实上，笔者也没有完全遵守匈牙利命名法，在很多场合，笔者往往在找不到标准的情况下，会自定义一个命名原则来使用。请大家注意，仅仅是为了表意方便。

3.3.2 函数命名原则

笔者的函数命名，一般严格遵守匈牙利命名法，即全部英文（严禁汉语拼音），首字母大写。不过，笔者根据经验，还有如下补充：

1、每个单词尽量选择简单的英文单词，强调表意性，力图在阅读者英文水平很低的情况下也看得懂。

2、不怕命名很长，强调函数名尽量表意，甚至考虑将算法直接说明在命名中，以达到一目了然的结果。

3、函数书写要完整，无参数的函数，要在括号内显式声明 void，每个参数也要符合变量命名法则。

在笔者的程序生涯中，一般会将函数名和变量名等定义得很长，最长的一次，笔者一个函数名有 237 个字符，这受到很多网友的非议，反对派最大的理由就是书写和观察不便。

不过笔者不这么认为，因为笔者在书写程序时，有个习惯，“**每个名字只写一次，以后全部拷贝**”，因为笔者认为，人是会犯错误的，再简单的名字，也有键入时笔误的可能，与其每次使用都重新输入一遍，不如在一开始定义好名称后，以后全部拷贝，因为计算机是不会出错的。

这在 VC++ 的 IDE 编辑器中，可以简单地归纳为 Ctrl-C + Ctrl-V 两个按键来完成。因此，再长的命名，笔者从来不认为输入是个问题。

另外，关于长命名不好看，太长的，超过一个屏幕的，确实很难以阅读，不过，大多数时候，一个命名还是很难超出屏幕范围的，反而由于名字很长，容纳的信息够多，使阅读者直接从名字上就可能看到很多信息，这比来回翻查程序段落，每句话都要通过阅读程序和注释来理解要好得多。

以下是笔者常用的函数命名例子，各位读者可以体会一下：

```
//一个弹出式缓冲区的清空函数
void CTonyPopBuffer::Clean(void);
//弹出式缓冲区的服务函数，以 bool 量表示内部的各种资源申请是否完成，是否可以服务
```

```
bool CTonyPopBuffer::ICanWork(void);
//安全的字符串拷贝函数
void SafeStrcpy(char *pDest, char *pSource,int nCount);
//安全的字符串缓冲区打印函数, 需要输入缓冲区指针, 限制长度, 变参打印
int SafePrintf(char* szBuf,int nMaxLength,char *szFormat, ...);
```

3.3.3 变量命名原则

变量名在匈牙利命名法中已经有了详细的定义, 该命名法规定, 每个变量名必须以小写的缩写前缀, 显式说明该变量的类型, 如 `nCount`, 表示整型的统计变量, `pBuffer`, 表示指针型的缓冲区地址等等。并且, 对于不同作用域的变量, 如类成员变量和全局变量, 必须以显式的 `m_` 和 `g_` 前缀区分。

不过, 这也是匈牙利命名法最为诟病之处。笔者在网上遇到的最强烈的反对意见就是针对这一特性。

反对者认为这样会导致程序修改不便, 一个变量, 一旦需要改变使用地点, 如从类成员变量改为全局变量, 或者改变类型属性, 如从整型改变为 `double` 型, 则必须显式修改变量名, 同时修改程序中每一个使用这个变量的地方。这给程序员带来极大不便。

但笔者认为, 这恰恰是匈牙利命名法最具有优势之处。我们知道, 在一个严谨的商用工程系统设计中, 关键变量的定义一般都是在系统设计阶段已经基本确立好了, 其作用域, 类型都应该在系统设计文档中做严格规定, 这类变量一般不允许在工程实施过程中再改变定义, 按照匈牙利命名法, 其定名也不应该改变。

因此, 上述的修改在严谨的工程实战中, 本身就是不合法的。如果出现这种情况, 恐怕项目组要开会, 重新讨论系统设计的合理性了, 而不仅仅是某个程序员简单修改变量名那么简单的事情。

其次, 退一万步讲, 即使是局部变量, 不需要提升到系统高度去讨论, 但一个变量在开发过程中, 突然改变了作用域, 或者类型, 程序员无论如何都应该仔细检查每个调用点, 看这个修改是否合理, 是否会引发新的 bug。

而匈牙利命名法强迫的变量名改变原则, 一旦变量类型改变, 其命名随之改变, 然后会引发编译器报错, 这样可以让编译器帮助程序员找到所有引用地点, 迅速定位, 检查潜在的 bug, 这比程序员凭肉眼人工检索要准确得多。

这种“强迫检查”的特性, 是笔者最看重的。

当然, 变量命名, 范围很广, 很难几句话说明清楚, 笔者一般在使用变量是, 有如下一些自我约定:

- 1、变量名表意清晰, 基本按照匈牙利命名法原则实施。

- 2、变量在函数使用过程中, 严禁转义, 即不允许一个变量在两个程序逻辑中作为两个含义的变量使用, 哪怕这是同一个函数内部。

- 3、函数内部变量, 原则上不加 `m_` 前缀, 直接以类型缩写开始, 如 `nCount`, `pBuffer`, 对于约定俗成的循环变量, 可以使用 `i`, `j`, `k` 来命名, 对于约定俗成的统计变量, 可以使用 `m`, `n` 来命名, 但仅限于此。

- 4、`sz` 前缀的指针, 表示静态数组指针, 不需要后续程序释放, `p` 前缀的指针, 表示动态申请指针, 需要后续程序释放。个别类或结构体指针的别名, 以 `p` 为前缀, 但并不需要后续程序释放。

- 5、结构体内部成员变量, 和类一样, 用 `m_` 作为前缀。

下面是笔者常用的变量命名, 各位读者可以体会一下:

```
//结构体内部成员变量, 也以 m_ 作为前缀
typedef struct _TONY_XIAO_MEMORY_REGISTER_
```

```

{
    void* m_pPoint;        //这是动态指针，需要释放
    char m_szInfo[TONY_MEMORY_BLOCK_INFO_MAX_SIZE]; //这是数组，不需要释放
}STonyXiaoMemoryRegister;
CTonyLowDebug* m_pDebug;    //类成员对象指针，需要释放
CMutexLock m_Lock;          //锁对象实例，m_之后直接是变量名，没有指针前缀
//结构体数组实例，m_之后直接是变量名，没有指针前缀
STonyXiaoMemoryRegister m_RegisterArray[TONY_MEMORY_REGISTER_MAX];
int m_nUseMax;               //整型数字，使用的最大数字（字面理解）
void* m_pMaxPoint;           //统计用，使用的最大指针，从字面理解，这个不需要释放
int m_nPointCount;          //所有指针的统计和，整型

```

笔者在带领团队的过程中，一般要求团队成员按照匈牙利命名法来定义变量，这样很多情况下，Review 开会项目成员的代码，基本上可以一目了然。

比如一个程序员曾经问过笔者下面一段程序有什么 bug

```

void Func(void)
{
    CInfo m_pInfo;
    m_pInfo.Print(".....");
    //...
}

```

这段代码，笔者一看就很别扭，函数内部变量，使用 m_前缀，命名是对象实例，缺定义了个 p 前缀，给人感觉是指针，同时，又实用 “.” 来调用内部函数，而不是使用指针的 “->”，总之怪异得很。

笔者当时让他把程序拿回去重写，必须严格按照匈牙利命名法书写，结果写完后，他向笔者报告，bug 已经自动解除了，这完全就是一次命名法不严谨导致的书写错误。

当一个程序员养成一个良好的命名习惯，对于程序就有了一种天生的敏感性，此时再去看没有严格规范的代码，就好比让一个人来看一篇充满了错别字的文章，直觉上就觉得很难受。这种代码规范敏感度，在开发几万，十几万行代码的大型商用工程时，是非常必要的。

最后，笔者还是简单列举一下匈牙利命名法常用的变量前缀，帮助读者进行后续的阅读。

```

n    int
c    char
b    bool
sz   静态数组，内存区，不可释放，可以当指针用。
p    动态指针，可能会被释放。
us   unsigned short
ui   unsigned int
ul   unsigned long
d    double（唯一允许被使用的浮点数类型）

```

3.3.4 其他命名规则

除了函数名和变量名外，C 和 C++ 程序书写时，也还有很多元素需要关注，下面笔者做简要说明。

3.3.4.1 类定名

针对类的定义，微软的 MSDN 已经给出了比较标准的定义，即大写 C 作为前缀。如下例：

```

class CmutexLock    //以 C 为前缀，定义类名，以后用此名定义变量
{
public:
    CMutexLock(void);        //构造函数
    ~CMutexLock(void);      //析构函数
public:
    void Lock(void);         //成员函数，直接表意
    void Unlock(void);
private:
    MUTEX m_Lock;            //内部成员变量，私有，且 m_前缀
};

```

3.3.4.2 结构体定名

结构体的定义方式，一般没有明确规定，笔者喜欢写成如下格式：

```

//每次结构体的定义，均以 typedef 显式定义出一个类型，以便用于后续的变量命名。
//以表意方式，加下划线，来写出结构体的真实名。_CListen_ListenAcceptTask_Param_
typedef struct _CListen_ListenAcceptTask_Param_
{
    Linux_Win_SOCKET m_nSocket;    //内部成员以 m_前缀
    CListen* m_pThis;              //顾名思义，指向本对象的指针，相当于 this
}SCListenAcceptTaskParam;         //以 S 定义结构体类型名，以后定义变量使用这个名字
//习惯性的，每写完一个结构体，都定义一个以 Size 为后缀的长度，方便后续的内存申请动作。
const ULONG SCListenAcceptTaskParamSize=sizeof(SCListenAcceptTaskParam);
//原则上，结构体内部不允许使用成员函数，如果有必要，请书写成类的格式。结构体在笔者的程序中，
//仅用于纯数据的表示，也就是纯 C 的定义。
//上述定义使用时，更加类似于类的使用方式
SCListenAcceptTaskParam ListenAcceptTaskParam; //直接定义实例
SCListenAcceptTaskParam* pParam                //定义指针
=( SCListenAcceptTaskParam*)malloc(           //指针类型转换
SCListenAcceptTaskParamSize);                 //如前述，有了长度定义，可以直接申请内存

```

3.3.4.3 宏定名

C 和 C++中，宏的定义分为两类。一类是常量宏，在 C++中，通常建议使用 `const` 定义常量来代替。不过，由于有时候工程代码需要引入一些开源或者古老的代码，这个宏还是有必要存在的。笔者在书写时，原则上是表意的单词加下划线，并尽量大写。

`#define USER_ID_LENGTH (256)` //定义用户 ID 的长度为 256 字节

另一类是函数型宏，这在笔者的开发中使用较多，一般在进行一个计算公式的确定时，笔者喜欢使用函数型宏，将正确的计算逻辑固化下来，这很类似于函数式编程。

```

//一个缓冲区元素的长度计算公式
#define TONY_POP_BUFFER_TOKEN_LENGTH(n) \
(n+STonyPopBufferTokenHeadSize)    //使用宏来固化算法
//一个元素的数据区开始点计算公式
#define TONY_POP_BUFFER_TOKEN_DATA_BEGIN(p) \
(p+STonyPopBufferTokenHeadSize)

```

还有一类函数型宏，笔者也比较喜欢使用，即某段代码的调用逻辑太过于复杂，后续代码很容易写错，笔者通常使用一个宏来固化调用逻辑，避免后续因笔误导致 bug。

```

//这段代码定义以加锁方式 call 某个函数，最后返回 bool 变量的过程，锁定名为 m_Lock
#define TONY_BOOL_LOCK_CALL(func) \
{ \
    bool bRet; \

```

```

    m_Lock.Lock(); \
    bRet=func; \
    m_Lock.Unlock(); \
    return bRet; \
}
后续逻辑当希望以加锁方式运行某个函数时，可以使用如下两种写法，都有效：
//第一种写法，写出完整加锁调用逻辑
bool FuncWithLock(void)
{
    bool bRet=false;
    m_Lock.Lock();
    bRet=Func();
    m_Lock.Unlock();
    return bRet;
}
//第二种写法，利用函数型宏，简化书写，而且还不容易有笔误
bool FuncWithLock(void){TONY_BOOL_LOCK_CALL(Func());}

```

3.3.4.4 标签定名

在笔者的程序书写中，很多时候都使用了 `goto` 语句，这个原因在后文有详细说明。但我们必须关注，C 语言的 `goto`，必须显式定义标签，这里，笔者也说明一下标签的定名原则。

笔者常用的标签定义原则，就是“类名+下划线+函数名+下划线+用途”。

常见的如：

```

CTonyRelayProtocol_CreateUpTunnel_End_Process:    //函数结束跳转点
CTonyRelayProtocol_CreateUpTunnel_Error_Process:  //函数错误跳转点
CTonyRelayProtocol_CreateUpTunnel_Loop_Next:      //continue 点
CTonyRelayProtocol_CreateUpTunnel_Loop_End:       //break 点

```

由于 C 和 C++ 语言，标签在编译时一般都具有全局性，因此，一定不能重名，笔者的这个定名原则可以有效防止重名 bug。

3.3.5 定名的折中

虽然我们在本章说明了很多定名原则，但是，也有一些代码管理相关的细节，我们必须关注。

在很多商业公司中，由于资源所限，通常无法安排专人管理代码，而是由各个程序员自行维护各自的代码库，以期在以后的工作中，实现重用。这是大多数中小型商业公司的现状，我们必须尊重。

但我们知道，代码重用，就表示一段成熟的代码模块，很可能被用到几个商用工程中，这就带来了一个定名的重名问题。

由于代码模块平时都是由各个程序员开发，在其自己维护的程序中，一般不存在重名问题，但如果有一天，这段代码与其他程序员的代码，整合到一个新工程中，如果定名原则中，没有注意区别，则很可能因模块重名导致编译失败。

在这种情况下，程序员被迫修改成熟的工程代码，很可能引入新的 bug，影响项目的进度。这在很多通用性模块命名中，表现尤为严重。

例如，程序员 A 和 B，在过去的工程中，各自维护了一个 CBuffer 模块，用来描述二进制数据缓冲区，并各自为 CA 和 CB 两个类服务。

这种设计很常见。但我们知道，Buffer 是个通用性很高的词汇，很多地方都用到，因此，一旦某一天，由于一个新工程需要，我们需要同时调用 CA 和 CB 两个模块时，大家发现什么问题没有？我们拥有了两个 CBuffer 类，名字一样，但内容很可能不一样。这在新工程的整合中，直接造成的就是编译错误，命名重定义，无法连接。

这时候，处于“尊重成熟代码”原则，我们一般不建议修改 CA 和 CB，使用统一的 CBuffer，避免引入新的 bug。因此，就必须修改两个 CBuffer 的命名，因为根据经验，命名修改可能出现的 bug，总是比逻辑修改要少一些。

这种情况笔者在过去的工程中遇到过很多次，每次出现都使笔者非常痛苦。因此，笔者一般建议，程序员在完成模块时，各种定名越长越好，尽量加上原有使用环境的修饰词，比如，CA 是一个传输模块，它使用的 CBuffer 建议就定名为 CABuffer，相应的，CB 就是 CABuffer，这是为了以后的重用降低重名率。

不过，也有很多时候，模块本来的功能就是通用性功能，很难加上特定的修饰，就如本文中 CBuffer 的定名，二进制缓冲区，本来就应该这么定名，加上环境修饰，就修改其意义了，会给以后的使用者误导。

这时候，笔者通常有个不是建议的建议，就是建议以程序员的英文名，中文名拼音缩写，作为该模块的修饰词，至少，我们知道，一个程序员一般不会无缘无故写两个功能完全一样的模块，因此，在他的模块中，这个 CBuffer 应该只有一份，不会重名。

比如笔者的英文名是 Tony.Xiao，一般笔者很多模块，习惯性地添加 Tony 修饰词，比如，CTonyBuffer，这表示是我这个人写的二进制缓冲区类，又或者，以笔者的姓名中文拼音缩写来修饰，如 XGSyslog，这表示是笔者这个人写的系统日志函数，以此和别的程序员的定名做区分。在后续的工程代码库中，大家可以看到很多这种设计，特别是一些底层的，重用性很高的代码模块，基本都使用了这种定名方法。

用这个方法，笔者的代码模块在和其他程序员的工程整合时，从来没有出现重名错误，因此，虽然这个方法有点“和稀泥”的感觉，并不算一个很好的建议，不过，笔者建议各位读者在以后的工程项目实施中，特别是多人合作的项目实战中，不妨采用一下，也许能减少很多不必要的人为 bug。这算一种定名的折中原则。

特别需要注明的是：这种方法并没有什么理论依据，甚至看起来很不正规，在很多学院派老师的眼中，这基本上属于大逆不道的定名了，在很多严格的企业研发管理人员看来，这也是给公司代码加上了程序员的个性标记，不是很职业化的行为，应该予以禁止。不过，笔者认为，程序设计，是一门实践性很强的学科，实践出真知，很多时候，理论上未必说得通，但实际用起来效果好的方法，我们也不妨一试，“**不管黑猫白猫，只要抓得到耗子就是好猫**”，因此，在某些条件有限的场合，大家不妨试一试，以程序员个性化的名字来修饰代码模块定名。

3.4 无错化程序的基本书写原则

前文我们说了很多基本原则，实现无错化的程序的书写，还是有一定原则的。下面就是一些笔者长期坚持的原则，对笔者的程序生涯帮助很大。

3.4.1 写简单易懂的程序

笔者在带团队进行开发的时候，曾经无数次的强调，程序开发一定要够简单，最理想的程序，由很多模块或函数段落构成，而每个模块都非常简单。

但很可惜，笔者碰到的大多数程序员，都很不注意程序模块的规模，几百行，上千行的一个函数比比皆是，然后测试，一堆的 bug，最要命的是，还找不到 bug 在哪里。最后笔

者没有办法，强行规定，每段函数，除非经过评审，认为必须写那么长，否则不允许超过 100 行，一般都控制在 20~30 行左右。这样下来，才把项目的 bug 率控制住。

前文强调了定名的重要性，其实笔者很多时候，把函数的构建，看做一小段程序逻辑的定名过程，当然，我们知道如果逻辑太复杂，这个定名就太复杂了，因为表达的意思太多了，最后，长函数名就出来了。

反而是如果函数够简单，比如就做个简单的 $a+b$ ，我们就可以简单实用 Add 来定名，至于它是什么的 Add，根据类命名空间的上下文，也很好判断。因此，严格的定名，其实也是逼着程序员写简单的代码。

所以，笔者在工程实做时，每写一小段逻辑，就立即用一个函数封起来，笔者看做这是段落大意的总结，大纲性的东西，或者说，按照人的自然语法，包含主语、谓语、宾语的一句话，可以作为一段话的一个成员，最后，所有这些话连到一起说，就是一片文章，一段程序。

比如下面的程序段落示例：

```
bool CTonyRelayProtocol::ICanWork(void)    //我可不可以工作，顾名思义，判断初始化成果的函数
{
    //... 严格判断，所有条件符合，方才返回 true
    return true;
}
//发送命令函数
int CTonyRelayProtocol::SendCmd(Linux_Win_SOCKET s,char cCmd)
{
    if(!ICanWork()) return 0;                //如果我不能工作，返回 0
    //...
}
//发送 ID 的函数
int CTonyRelayProtocol::SendID(Linux_Win_SOCKET s,char* szID)
{
    if(!ICanWork()) return -1;               //如果我不能工作，返回-1
    //...
}
```

大家可以看到，笔者把判断本类对象初始化是否完成这个工作，通过一个函数，定名为 ICanWork，这样，后续所有的程序写出来，都有一句 if(!ICanWork())...的判断语句，这既便于理解含义，也起到了防护设计的作用，而且，后续的函数长度，也都获得了控制，效果很好。

请各位读者注意：随时随地关注你的程序是不是够简单，笔者经常和团队伙伴说的一句话：“如果你的程序写得不够简单，就已经写错了！”。

简单程序设计的原则如下：

1、每个函数，只写一个独立的逻辑段落，其逻辑复杂度以英文一句话能说清楚为原则，并且，这句话就是函数名。

2、这表示，函数中，一般说来，只能有一个循环体，内部可以是多重，但并列的只能有一个，不能有两个以上并列的循环体，根据经验，这一般都是多个逻辑了。

3、同理，一个函数中，尽量只能有一个和业务相关的完整的判断逻辑，即 if...else...逻辑，可以内嵌多重，但不要有并列。当然，出于防御性考虑设计的参数判断不在此列。

4、同理，一个函数中，尽量只能有一个完整的 switch 结构，不要并列多个，且，switch 结构内部应该全部是 call 调用，原则上不应该有具体的执行语句，这是避免 switch 所在的函数逻辑太丰富，导致 bug。

5、原则上，每个函数内部的大括号层数不要超过3层，这是因为据笔者所知，至少 gcc 的编译器，对于超过3重的大括号程序块，拒绝优化，如果超过，请将内部的模块摘录成一个独立函数调用。（类私有函数个数不怕多）

6、一个函数原则上不要超过20~30行，最长不建议超过100行，超出可以考虑强行分界，定义为多个函数。尤其是“实现功能”的复杂代码，必须遵守这一原则。

7、函数的返回值尽量简单，能返回 bool，绝对不返回 int，能不返回，就坚决不要返回，函数的参数设计精确，定名表意性好，尽量简化后续调用者的理解学习过程。

8、一个类包含的数据种类原则上只能有1~2种，这是因为对象的含义就是一笔数据以及其相关方法的集合，数据种类太多，类对象做了太多的事情，逻辑就可能会非常复杂，导致 bug。太大的类，考虑内部进一步抽象更小的类来分割包容。

9、结构体主要用于参数传递，不要包含成员函数，这样，在偶尔用到联合体的时候，不至于发生不必要的 bug。

简单程序设计非常重要，可以说是无错化程序设计的核心，笔者在这里摘录一段在以前发表的散文《简简单单写程序》，请大家体会一下，简单开发的深刻含义：

写程序这么多年，总有几百万行了。感觉，程序写的方向，总的来说越来越简单。

一个函数，简简单单几行就完了，一个类，简简单单几个方法，也就完了。

简单，就好懂，自己看自己的代码，看昏了的也有，一个函数几千行，人的脑子，堆栈显然不够用，看了后面，忘了前面，何苦呢。

简单，就不容易出错，一个工程完了，心里其实很发抖的，很怕测试报 bug。不过，程序简单，每句话都一目了然，自己看了，每段话的意思都说清楚了，它就不可能出错。除非写少了，或者写多了，但只要写的，就是对的，他们凭什么报 bug。心理平安，晚上回家，才可以愉快地喝点小酒，晕个小觉。人生愉快啊！

简单，就是问题弄清楚了，需求明确了，方案有可实现性。客户的一个需求过来，基本上都是笼统的，模糊的，大的吓死人，什么时候，分析清楚了，什么时候，才可能把功能细分到很简单的地步，能写简单的代码，其实是系统分析成功的结果，是一种幸福。

简单，就是赏心悦目，一段程序，和一篇文章一样，太复杂，总是不好的。简简单单，计算机轻松，人也轻松，哪怕过十年看回去，也会忍不住自得一下，很愉快。这世上，又有多少人，看自己十年前的文章，不后悔、不遗憾、不会感到不好意思？

简单，就是轻松，自己的程序，别人好懂，别人的程序，自己好懂，沟通起来方便，谈起问题来愉快，大家天天开开心心做工作，何乐而不为？

简单，就是家庭幸福。程序员很苦的，无数的加班，无数的苦思，其实，说来说去，都是 bug 闹得。为啥有 bug 呢，程序写复杂了，自己看都头晕，计算机怎能不晕。简单一点，bug 就少一点，加班就少一点，最终多点时间陪陪家人，谁说程序员不能享受天伦之乐？

简简单单写程序！

简简单单过生活！

程序员的世界，简约，但是不简单！

3.4.2 严禁变量转义

这实际上是程序简化的补充，很多程序员，由于程序写得很长，嵌入了太过于复杂的逻辑，涉及了大量的变量，很多时候，程序员出于偷懒等目的，就在后面程序段落沿用前面用过的变量，结果导致 bug。

笔者开发的商用工程，对变量转义是严厉禁止的，因为这种用法通常都会出 bug，而且 bug 很不好查找。

我们很多程序员都知道，变量声明后马上赋初值，避免使用“野变量”，变量转义本身就可以看做野变量，如果后续代码不做显式赋值，变量的初值完全取决于前面逻辑的计算。非常危险。

所以，变量转义和复杂程序，可以说是孪生兄弟，往往出现在一起，如果程序员稍有不注意，就可能会导致无法查找的 bug。

控制变量转义，简单说，就是每个变量，通过严格定名，在其作用域内有且仅有唯一定义，仅做一个用途，坚决不做它用。

请大家放心，根据笔者经验，就算是嵌入式设备等内存很小的应用场合，也不会因为多定义一个简单变量，而导致内存溢出。多定义几个变量，每个变量各司其职，会有好处的。

3.4.3 严禁一语多义

这实际上也是程序简单化的补充。一语多义，就是在一行程序中，容纳了太多的计算动作，过于复杂，导致调试不便，计算顺序不好判断，程序易读性不好，也容易引发 bug。

笔者在长期从事 C 语言的开发中，发现 C 语言提倡的一些书写习惯，确实很不好，特别是一些缩写，一语多义的情况，很多 C 程序员热衷于此，并且还津津乐道。给商用工程带来了严重的 bug 隐患。更为严重的是，很多程序员在转向 C++ 之后，仍然保留了这些陋习，导致 C++ 这门比较严谨的语言，在开发上质量也不高。

在很古老的 UNIX 时代，由于集成电路工艺的限制，内存还是一个很奢侈的东西，很昂贵。记得笔者 1993 年购买的 4M 内存条，花了差不多 1280 元人民币，再早些时候，很多大型机的内存，可能也仅有 128k 而已。

这直接造成了操作系统的内存紧张，并直接带来了对程序员节约内存空间的开发需求。不仅仅要节约运行空间，连程序员的程序都要尽量短小精干。变量名用缩写，表意性不好的习惯就是这个时期养成的。

笔者记得 86 年的一份《学生计算机世界》报，开辟专栏，引导大家写一程序，即在 AppleII 上，利用一行输入的极限，254 字节内，书写出一个完整的程序逻辑。笔者也尝试过，做了一个键盘控制的绘图程序。

但这给程序界带来了很坏的一个误导，这一时期的程序员，包括笔者自己，都曾经很喜欢使用缩写，并强调“一程序”，于是，一语多义的现象大量出现，程序易读性变得很坏。如下面的例子：

C 的这个“(a)>(b)?(a):(b)”结构，就是典型的一语多义，明明可以写成 if...else... 结构，以缩进形式明确书写，但写成一行，不是很熟悉 C 的程序员，看起来很困难。

(a)>(b)?(a):(b)

再如这个例子

```
#define Max(a,b) (a)>(b)?(a):(b)
```

```
Max(++a,b+2)
```

明明可以简单写为如下格式：

```
a++;
```

```
b+=2;
```

```
Max(a,b)
```

结果写成一行，一行中有三个计算，并且使用了不可单步调试跟踪的宏，导致程序逻辑看不清楚，各个计算的先后顺序很费解，容易导致 bug，并且不易读。

在无错化程序设计方法中，笔者认为程序的易读性是程序本身的质量保证，并不是可有可无的东西，一语多义的现象，除了让程序员显得很“酷”外，对商用工程开发没有任

何好处，反而人为增加沟通成本，增加很多 bug 隐患，因此，笔者在商用工程中，一般严禁一语多义，要求每行程序只写一个计算。

3.4.4 函数只能有一个出口

这也是一个大家忽视的话题。在很多书籍中，讲了大量编程规范的重要性和方法，但笔者很少看到关于这一点的论述，这直接导致众多的程序员书写代码随心所欲，任意跳出功能逻辑，而根据笔者经验，在开发商用数据传输工程，甚至就是开发任何一个 C 语言程序中，这都是很致命的忽视。

与之对应，不止是 C 语言，几乎所有的程序设计语言，在设计之初，都教大家可以从循环内部跳出，函数内部跳出，但是不允许从外界跳入。这也无形中误导了很多程序员，认为从逻辑模块中跳出是合理又合法的。这进一步加深了这种失误。

在很多语言中，不止是 C 或 C++，都有二元性动作，即语句两两配对，缺一不可，共同完成功能。

举几个例子：

C 语言的内存申请动作，malloc 和 free 就是二元性动作。

C++语言，其他一些面向对象语言的 new、delete，也是二元性动作。

所有的语言，只要支持多线程，其加锁和解锁的过程，也是二元性动作。

还有 socket 传输中，产生一个 socket 和关闭一个 socket，也是二元性动作。

大家都知道，针对这类二元性动作，一般都遵循“谁分配，谁释放”的原则，或者“谁进入，谁退出”原则。就是一个函数，或者对象，做了一次资源申请，或者加锁动作，一定要在本函数或对象退出时，负责释放，这是程序无 bug 的关键。笔者在前文论述的资源锁概念，就是这一原则的具体体现。

针对对象，一般比较简单，在构造函数内分配，在析构函数内释放即可。但针对函数，就非常麻烦。原因很简单，函数的退出机制太多了，程序员完全可以通过一次 return，跳出所有后面语句，实现退出。当然，大多数时候，也就跳过了二元性动作的第二次动作，于是，bug 就产生了。

因此，笔者在带领团队开发商用工程时，最强调的一件事情，就是“函数只准有一个出口”。不仅仅是函数，所有的循环，不管是使用 break 跳出，或者使用 continue 提前结束本轮循环，都有且仅有一个点，避免类似的 bug。

下面是一些程序段落的例子，大家可以看看这类 bug 的危害，以及预防方法。

1、内存泄漏的出现和防止

```
void Func(void)
{
    char* pBuffer=(char*)malloc(100);
    if(...) return;    //此处提前跳出，导致后面 free 没有执行，内存泄漏
    //...
    free(pBuffer);
}

void Func(void)
{
    char* pBuffer=(char*)malloc(100);
    if(...)            //此处跳至最后退出点，确保 free 执行，防止内存泄漏
        goto Func_End_Process;
    //...
Func_End_Process:
    free(pBuffer);
}
```

```

}
2、循环跳出导致的问题
void Func(void)
{
    bool bContinue=false;
    bool bBreak=false;
    char* pBuffer=null;
    while(1)
    {
        pBuffer=(char*)malloc(100);
        //...
        if(bContinue)    //此处提前终止本轮循环，导致后面 free 没有执行，内存泄漏
            continue;
        if(bBreak) break;    //此处跳出循环，导致后面 free 没有执行，内存泄漏
        //...
        free(pBuffer);
        pBuffer=null;
    }
}

void Func(void)
{
    bool bContinue=false;
    bool bBreak=false;
    char* pBuffer=null;
    while(1)
    {
        pBuffer=(char*)malloc(100);
        if(bContinue)    //此处跳至循环继续处，后面 free 执行，内存泄漏
            goto Func_Loop_Continue;
        if(bBreak)    //此处跳至循环结束处，
            goto Func_Loop_End;
Func_Loop_Continue:
        free(pBuffer);
        pBuffer=null;
    }
Func_Loop_End:
    if(pBuffer) //此处保证中途跳出，pBuffer 能释放
    {
        //注意，前面 pBuffer 赋初值 null 的含义在此体现
        free(pBuffer);
        pBuffer=null;
    }
    return;
}

```

还有些特例，虽然不是二元性动作，但也必须要关注，比如前文我们讨论时间片的时候，强调在每个循环中使用 **Sleep** 释放时间片，这虽然不是二元性动作，但在线程中，这是每轮循环必须完成的任务，因此，也必须跳转强迫执行。

举个例子，一次一个同事问笔者一个问题，说我们的 Asterisk 服务器，他在修改里面一段代码，但出现一个奇怪的现象，两个人通话时，CPU 占用率正常，反而一方不说话时，没有数据传送时，CPU 占用率暴涨到 50%左右，这是怎么回事？

笔者想了一下，写了如下一段逻辑，让他去程序中看，是不是这么写的：

```
while(1)
{
    if(NoSound()) continue;    //如果没有声音，则 continue 结束本次循环
    //...                      //注意，Sleep 没有被执行，导致 CPU 占用率过高
    Sleep(1);
}
```

他一查，果然如此，这就是典型的循环中途结束，导致最后的 Sleep 没有被执行导致的。笔者一说，他自然知道怎么修改了。应该修改成：

```
while(1)
{
    if(NoSound()) goto Label;  //如果没有声音，强制跳转到 Label，
                                //Sleep 后结束循环，CPU 占用率恢复正常
    //...
Label:
    Sleep(1);
}
```

二元性动作，在 C 和 C++语言的开发中，以及多任务操作系统的开发中，几乎无处不在，很多程序员学习多线程编程，遇到困难，很大一个原因就是对于锁的使用，资源的使用，没有高度的警惕性和严格的控制，函数书写随心所欲，跳出点众多，导致大量的 bug。因此，本小节的内容，请各位读者一定要关注，很多时候，这是解决 bug 的关键。

3.4.5 变量如不使用，保持初值

这算是野指针问题的延伸。前面我们讲过，野指针是内存错误的根源，没有赋初值的指针是非常危险的。由此引申，野变量也是很危险的，因为其初值无法确定，完全取决于程序运行的中间情况。如果一段程序逻辑，对上文的变量没有赋初值而使用，其结果是不确定的，可能正确，也可能错误。

比如这个例子，由于 i 值不确定，因此，循环次数不可知。

```
void Func(void)
{
    int i;
    for(;i<100;i++) //请问这个循环有多少次？
        //...
}
```

同理，如果一个变量，即使在声明时已经赋过初值，在前段逻辑中已经使用过，初值已经被破坏，而后文继续使用，如果不是业务需要这种数据的传递，则很可能导致 bug。

//还是上文的例子，不过修改一下，大家请看：

```
void Func(void)
{
    int i=0;
    for(i=0;i<15;i++)
        //...
    //此处 i 已经变为 15
    for(;i<100;i++) //如果下面的循环希望是 100 次，则由于 i 有残值，逻辑上已经出现 bug
        //...
}
```

这其实也解释了前文 5.3.1 节的原理，为什么要写简单程序，每个函数建议只有一个循环体，可嵌套，不可并列的原因。

任何一个函数中变量，理论上如果不是在用状态，应该及时赋回初值，这是保证程序安全的不二法门。最典型的例子，就是下面的指针使用。

```
void Func(void)
{
    char* pBuffer=null;                //声明就赋初值
    pBuffer=(char*)malloc(100);        //此处开始使用
    if(!pBuffer) goto Func_End_Process; //防御性设计（跳转点）
    while(1)
    {
        if(...) goto Func_End_Process; //条件退出循环（跳转点）
        memcpy(pBuffer,...);           //使用 pBuffer
        if(...)
        {
            free(pBuffer);              //某种情况下，可能业务逻辑需要清空该指针
            pBuffer=null;                //注意此处的赋回初值。
        }
    }
}
Func_End_Process:
//请注意前面的段落，有两个点会跳到此处
//跳至此处时，pBuffer 可能是有效的，也可能是无效的
//如果前面的 free 不赋回初值，则下面这个判断将不准确，出现 bug
if(pBuffer)
{
    free(pBuffer);                      //因此，指针任何一次清空
    pBuffer=null;                       //习惯性地立即赋回初值 null
}
}
```

上述样例，在商用工程中很常见。由于要不断根据新到的数据长度，调整缓冲区大小，因此，指针一般都是不断在刷新的，而由于某个突然的异常，导致程序退出时，需要根据情况释放指针，此时的赋回初值，就变成了指针有效性的唯一标志，可确保无 bug。

3.4.5 常量必须定名

这也是很重要的一条。在笔者审核很多程序员的代码时，无数次发现一些很奇怪的常量，没有前因后果，很突兀地就出现在程序中，让读者莫名其妙。如下面的例子：

```
void Func(const int nIndex)
{
    int i=0;
    for(i=50;i<=101;i++) //这个 50 和 101 是怎么回事？哪来的，为什么是这么多？
        //...
}
```

笔者经常说，某些程序员喜欢出谜语让别人猜，就是这个意思。

这是很不好的习惯，任何一次计算，其实理论上讲，都是有意义的，没人会去写无意义的代码，但是常量本身虽然有值，但其书写的过程中，并没有携带其含义，因此，常量太多的代码，基本不具备阅读性。

所以后来笔者就给程序员规定，所有的常量，必须用宏，或者 `const` 显式定名，这样程序中一目了然。如上例：

```
#define SEARCH_BEGIN 50
#define SEARCH_END 101
void Func(const int nIndex)
{
    int i=0;
    for(i= SEARCH_BEGIN;i<= SEARCH_END;i++)    //这是一次检索活动
        //...
}
```

基本上，笔者审核代码，如果常量没有定名，直接在程序中出现，都会打回去重写。只有下面几个例外：

- 1、常量可以出现在给变量赋初值的地方，但仍然要求注释说明其来源。
- 2、0 可以出现在 `for(i=0;i<n;i++)` 中，这是唯一可以在程序逻辑中使用 0 的地方。
- 3、1 可以出现在 `while(1)` 中，这也是唯一可以在程序逻辑中使用 1 的地方。
- 4、`null` 原则上仅用于赋初值，判断语句一般使用 `if(!p)` 来代替。
- 5、其他特殊情况，出现后，项目团队开会评审讨论。

最后，请各位读者注意，常量不仅仅是数字，程序中写死的字符串也是常量，也需要定名的。

3.4.6 太大数组不要用静态方式

在过去的商用工程的开发过程中，笔者考虑到系统一般都有 7*24 小时运营需求，需要规避内存碎片的产生，因此不建议使用动态内存分配，尽量以静态数组之类的结构完成计算。在大多数情况下，这种设计都是合理的，由于完全杜绝了动态内存的申请和释放，自然就没有内存泄漏和内存碎片的产生。

不过，在笔者前不久的一次服务器集群工程中，由于设计的服务器需要跨 Windows 和 Linux 平台通用，笔者的这个设计方法受到挑战，我们在设计中，需要使用一个 200k 的大数组，作为最大信令的缓冲区。但很不幸，在移植到 Linux 下时，程序崩溃，并且找不到原因。Windows 下工作一切正常。

最后我们经过长期分析，差不多有两个月，才发现原因是因为线程栈问题，每个操作系统，为一个线程规定的浮动栈空间是有限制的，虽然可以调整，但一般有一个默认值。Windows 下一般为 1M，Linux 下一般为 10M。

这就是说，不管这个线程层级调用多少层函数，所有这些函数的浮动栈加起来，不能超过这个限制，否则就出问题。而函数内部的静态数组，就是建立在这个浮动栈上的。

这里的 bug，根源在于，我们的服务模块，在 Windows 下是独立运行的进程，而在 Linux 下，是以动态链接库 so 形式，被 Asterisk 服务器调用，是在线程中调用。由于 Asterisk 服务器的线程，除了我们的服务外，还要处理一大堆其他事务，如语音软交换等，那些事务占用了大量的栈空间。

因此，尽管 Linux 下栈空间远大于 Windows，但我们实际可以使用的部分，不够 200k，函数进入后，还没有正式开始执行我们的业务代码，在初始化浮动栈静态变量时，已经崩溃，无法继续。

这对笔者是一个警告，静态数组虽然安全，但如果滥用大数组，对浮动栈的冲击会非常大，很多时候，会出现不可见的 bug 隐患，极其难以查找和消除。

因此，后来笔者改变了这一策略，专门做了一个内存池模块来实现动态内存块重用，规避了内存碎片，之后的开发，重新开始使用动态内存申请，这类 bug 自动解除。内存池在后文的工程库中有详细介绍，此处不再赘述。

3.4.7 尽量避免使用递归

递归在程序界一直是一个毁誉参半的设计，支持者强调递归使程序简洁，很多算法用递归开发会很容易；而反对者强调递归阅读费解，跟踪不易。各有各的道理。

在无错化程序设计方法中，笔者总的来说，还是不支持使用递归的。建议回避使用。

1、我们强调写简单易读的程序，有时候为了达到这个目的，甚至故意把可以一行写完的程序，分拆成多行，尽量使程序分段清晰，易读易懂。递归显然没有这个优势，程序员阅读递归程序，一般需要不断在脑中记忆递归的层数，每个判断条件的当前值，这些记忆随着递归层数的增加而增加，很容易超过程序员的记忆能力，最终导致看不清楚。

2、递归基本上没有办法单步跟踪，因为递归程序看似简单，却不断在同一段函数内进进出出，变量不断变化，递归通常又是复杂跳出条件的循环，一般又没有循环计数器，对于调试时理解程序的当前状态不利，程序员很难看清真实的调试状态。

3、递归成本很高，我们知道，任何一次函数调用，都会引发一系列的内存浮动栈建栈，返回点的记录过程，因此当使用递归表示一个循环关系的时候，其运行成本远高于一个循环。并且，C 和 C++ 编译器一般都没有递归尾部优化，即最后一个 `return` 直接跳出所有循环的设计，因此从运行态看，一个递归函数 `call` 进去多少层，`return` 回来就有多少层，实际运行时，相当于双倍的运行成本。

4、递归有隐含的 bug 隐患，由于递归的函数实际调用次数很多时候是看不清楚的，程序员很难精确计算一个递归逻辑的实际循环次数，而另一方面，递归是函数调用循环，如上文所述，每次函数调用，都会建立一些函数浮动栈，回调点，有一定内存开销，当递归层数多了，很容易因内存溢出导致程序崩溃。根据笔者经验，VC++ 的 debug 模式，一般 2000 重左右的递归，即可导致崩溃。这种崩溃最致命的是无法从代码上直观检查出来，一旦运行期发生崩溃，几乎无法查找 bug。

综上所述，笔者建议以后大家开发时，尽量减少递归的使用，除非能精确计算每种情况下的递归深度，除非算法必须递归，否则的话，建议尽量使用循环代替。

3.4.8 解决方案一套就够

在笔者经常关注的 CSDN 论坛中，笔者发现很多同学都在争论，一个逻辑，这么写好，还是那么写好，为此甚至争吵不休。有时候让笔者觉得很好笑。有时候看见大家争论，感觉好像大家正在为一条蛇，添加第 `n` 条腿用一样，纯属没有必要。

笔者在工作实际中，也遇到这类问题，同一个逻辑，很多写法，笔者承认，很多时候，软件性能优化，必须用到这个技巧，用不同的方法写出来的程序，效率就是不一样，不过，实际工作中，这些情况并不多。还有很多写法，完全和优化无关，仅仅是程序员的个人习惯，或者表现欲罢了。

比如后文笔者将要说明的循环语句，C 里面就有很多种写法，都对，但就这个问题而言，一个项目组，仅循环语句就有 7、8 种写法，其实是一种灾难，彼此很难沟通。

从维护工程库的角度，也应该强调这一思路，**商用代码，很多时候最有价值的，并不是代码本身，而是这个代码长期测试、运营之后的稳定性结果**，任何对代码的修改，替换，都可能造成新的不稳定，造成大量的成本浪费，因此一般说来，一旦一个算法被证明成熟，

并且入库，则以后类似场合，只要没有太大的性能差异，都应该无条件采用，不允许重新编写。

因此，针对程序书写中，与性能无关，与对错无关，仅仅是选左选右的问题，笔者一向强调：“**解决方案一套就够**”，太多了没有意义。简单说就是大家强行约定一种写法，都按照这个写法写，不要标新立异，用标准化开发保证工作效率，也提升程序的直观易读性，降低 bug 率。真的因为优化，需要特殊算法的，由项目组开会评审决定，这在后文大家可以慢慢体会。

3.5 基本程序设计原则

无错化设计方法，很多理念都脱胎于结构化程序设计，这里我们简单介绍一下程序模块的基本组织和设计原则。

3.5.1 函数的设计

函数的设计是 C 和 C++ 程序设计的基础，经过笔者总结，除了上述无错化程序设计的基本书写原则外，还应该关注的有如下几点：

1、参数设计越少越好，必要时考虑使用结构体包装参数表。原因很简单，参数少的函数，总是比较易读的，其次，结构体包装还有一个好处，就是调用者对参数定义非常明确，详情请参见后文使用结构体的技巧中的例子。

2、函数重载是一个 C++ 的特性，笔者个人认为是个很好的设计，一个函数，一旦被多次调用，会在程序员心中形成书写习惯，直觉上就知道怎么写正确，因此，几个同义的函数，仅仅是调用参数的类型不同，则尽量使用重载，使函数的返回值使用一套定义，这种对习惯的尊重，将有效保证调用者的工作效率。

3、尽量使用默认参数，在 C++ 的设计中，默认参数默认值也是个很好的设计，这相当于函数的设计者为调用者提供一个最佳调用建议，如果调用者心中不是很明确每个参数的含义和最佳值，可以简单使用默认，至少不会因此而出错。这是一个很明确的正确暗示，使函数的调用友好度大为增加。

例如：这是一个回调函数的示例，一般回调逻辑会协助调用者传递一个 void* 的指针，但有时候，回调函数的设计者，可能并不需要这个功能，此处，默认值表示可以忽略。

```
int FuncCallback(char* szData,int nDataLength,void* pParam=null);
```

4、返回值尽量简单，规范，这也是提升函数可用性，进而降低调用者学习负担的重要一点。前文已经说过，返回值设计，能赖则赖，尽量偷懒。笔者写一个函数，一般都是先写成 void，如果中途由于调用者需要状态返回，一般都返回个 bool 表示成功或失败，如果需要返回多态，则再考虑 int 型返回。

如果是 int 型，在商用工程中一般的原则是，返回操作的字节数，表示正确，返回 0 或 -1 表示异常，具体错误类型，根据业务定义来确定。比如 Send 的 0 和 -1 都是错误，而 Recv 返回 0，很可能是没有新的报文到来，线路本身不出错。

当返回值是几种状态代码，而不是有业务含义的数字时，一般需要使用宏或 const，显式定义每种状态常量的含义。给后续调用者一个清晰的提示，如下例，笔者的 recv 函数，需要返回几个显式的错误状态，采用如下定义：

```
#define SOCKET_RW_0_NO_ERR      0    //读写到字节,但是没有错误
#define SOCKET_RW_ERR_10054    -1    //10054 错误
#define SOCKET_RW_ERR_OTHER    -2    //其他类 socket 错误
```

5、函数内部尽量使用大括号，减小变量的作用域。这也是严禁变量转义的补充。如下例，程序员在某些场合，可以明确使用函数内大括号，显式区别变量的作用域，这不属于变量转义，非常安全。

```
void Func(void)
{
    {
        int i=0; //这两个 i 不是一个变量，因此不属于变量转义
        //...
    }
    //...
    {
        int i=0; //这两个 i 不是一个变量，因此不属于变量转义
        //...
    }
}
```

3.5.2 类的设计

类的设计，是 C++ 的基本功，根据笔者多年的经验，为了实现无错化代码设计，类的设计有如下注意事项：

3.5.2.1 只使用 public 和 private 定义。

protected 是 C++ 里面一个很奇怪的设计，本来“类”这个逻辑设计出来的目的就是为将数据私有化，因此 public 和 private 就基本够用了，但不知道为什么，非要弄出一个 protected，并且还有友元设计，让某些其他类可以看到自己的内部数据。对此，笔者一直不太理解，也从不使用。

除非是为了大规模的多重重载，以及程序设计得太复杂，笔者看不出 protected 有什么用途，而根据笔者的理念，一个类和一个函数类似，如果不能设计得够简单，不能够封装得够严密，就基本没有可重用性，因此，建议大家减少或取消对 protected 的使用，同时也不要用友元设计。简单建立一个 GetValue 接口，可能效果更好。

3.5.2.2 用聚合不用重载。

这可能是笔者有点偏激了，C++ 的一个重大优势，就是可以使用重载等面向对象的程序设计特性。不过，笔者一直记得一句话，是 C++ 之父说的，“**多重继承是场灾难!**”。

不同的类，有不同的特性，即使一个 C++ 很熟练的人，也很难从多重继承中精确判断出每个函数的视界，每个变量的作用域，当前是虚函数的第几层实现等信息，这会严重干扰程序的易读性，即使作者本人能理解并看懂，调用者只要水平稍微差一点，恐怕就是一头雾水，这个类对他来说，就是个“黑匣子”，这可能带来很多人造的 bug。

商用开发，不可能要求每个人水平都一样，也不可能要求每个人都是高手，毕竟，公司招聘人员，还要讲个配比，还要讲个综合成本。因此，在团队开发中，太复杂的设计，除了显示个别高手很酷之外，没有任何好处，最简单的永远是最实用的。因此，请尽量避免使用重载。改用聚合。

聚合就是原始的功能类，被使用者聚合为内部成员对象，通过二次 public 输出其公有方法，多重聚合，就是聚合多个内部成员对象。此时，程序员可以清晰地看清楚每个对象之间的相互关系，不容易出错。

另外，程序员作为人，总是很懒的，这个方法要求显式书写公有方法来输出功能，明确带来额外的工作量，那么，程序员很可能因为偷懒，故意减少公有方法的书写，这无形

中就保证了一个类尽量内聚，公有方法很少的结果，实现高内聚、低耦合的设计目的，保证产品质量。

很多人可能的反对意见，来自于这带来额外的工作量，可能会降低程序员效率，不过，现代的 IDE，如 VS2008，都有很强的代码补全功能，以及很多帮助书写功能，程序员对于重复、同名的代码，可以简单通过拷贝完成，根据笔者经验，聚合带来的工作量，并不大。

3.5.2.3 尽量不要用模板等 C++ 高端特性

这纯粹是经验之谈。笔者以前比较喜欢用模板，模板，就笔者的理解，就是“类的类”，其本质是封装算法，但不界定数据类型，当代入数据类型后，即可构建针对指定数据类型的类，进而实例化对象工作。

笔者在 2002 年时，曾经在 CSDN 上发表了《多线程安全的变量模板》一文，详细介绍如何利用资源锁概念实现多线程安全变量族。其中，主要就依靠了模板技术。但很不幸，当笔者在 2007 年左右进入跨平台开发时，发现一个问题，基于 arm9 嵌入式 Linux 平台的交叉编译器，不能很好地完成模板的编译。这个设计被迫废除。

经过分析，笔者发现 C++ 其实是一个成本很高的语言，其虚函数，多重继承，模板等高端特性，其实是依赖对内存的大量使用，建立表、栈来完成的。并且，出于通用性考虑，其内存浪费较为严重。

比如一个多次继承的类，某一个虚函数指针就是一个栈表，清晰列出其历史继承关系，调用者会调用栈顶的最后一次继承，并根据程序书写，决定是否逐一向下调用父类的虚函数。而所有这些内存占用，都是编译器根据程序书写的情况，编译时动态决定的。

嵌入式系统通常内存很小，一般都小于或等于 64M，加上操作系统的开销，能留给应用程序的运行空间并不大，因此，复杂的继承关系、模板的大量使用，会导致潜在的内存开销，这在编译时并不会表现，而在运行时，由于某个业务一次较大规模的内存申请，而导致失败，并引发程序崩溃。

我们强调，无错化程序设计方法的基本理念，就是数据边界严格可控，因此，大量对 C++ 复杂特性的利用，相当于对内存的无边界审核使用，类似于递归的结果，这类错误一旦引发 bug，首先是无法查找，其次是很难判断，并且修改成本也很高昂。

因此，一般的建议是，出于跨平台设计需求，建议尽量不要使用模板、继承、虚函数等 C++ 高阶特性。

3.5.2.4 随时注意向下聚合

C++ 是面向对象的程序设计语言，设计原则讲究高度的抽象性，即一个模块，一旦被多个逻辑调用，即考虑寻找其共性，抽象出一个通用的类或者函数，提供标准接口，避免每个逻辑各写一摊，以同一代码完成所有类似的功能。

因此，C++ 程序员的“向上抽象”思维都比较强烈，即如上所述，习惯着眼于一个功能对多个调用点之间的抽象。但笔者的工作中发现，由于上述思维，很多 C++ 程序员在另一方面反而有所薄弱，即“向下聚合”。

举个例子，如我们要写一个网络服务器，接到一个 socket 请求，响应，获得请求，计算，并回应报文，这是一个基本的工作逻辑。写起来也不难。一个类，用一个 socket 作为内部成员对象，写出相关接口函数即可。

```
class CAnswer
{
public:
    //...
    int Send(char* szData,int nDataLength)
    {
```

```

        //...
        return send(szData,nDataLength);    //单次操作业务
    }
private:
    Linux_Win_SOCKET m_nSocket;                //服务的 socket 资源
};

```

然后，新的需求来了，主管要求我们改写我们的代码，要应对至少 500 路的并发请求，一个没有什么经验的 C++ 程序员会怎么做，我们此处仅是简单示例，程序简单修改如下：

```

class CAnswer
{
public:
    //...
    int Send(char* szData,int nDataLength)
    {
        //...
        int i=0;
        for(i=0;i<500;i++)                    //循环操作业务
            send(szData,nDataLength);
        return 1;
    }
private:
    Linux_Win_SOCKET m_nSocket[500];          //服务的 socket 资源
};

```

但大家想过没有，如果这么设计，那么，我们再设计一个 Recv 的函数，是不是也要这个 500 次的大循环，其他的逻辑能，另外，根据业务不同，可能某个 socket 通道的信息要送到另外一个 socket，这个类的复杂度将会直线上升。

此时笔者提示，尽管只是个小小的需求改动，程序员也该考虑“向下聚合”了，即随时注意抽取一块具有通用性的逻辑，构建一个新子类，封装对数据的共有方法特性。

```

class CAnswerToken                                //将每个 socket 以及其动作封装为新子类
{
public:
    //...
    int Send(char* szData,int nDataLength) //提供标准方法
    {
        //...
        return send(szData,nDataLength);
    }
private:
    Linux_Win_SOCKET m_nSocket;                //服务的 socket 资源
};

class CAnswer
{
public:
    //...
    int Send(char* szData,int nDataLength)
    {
        //...
        int i=0;
        for(i=0;i<500;i++)                    //循环逻辑与 socket 无关，子类无需关心，父类处理

```

```

        m_Token[i].Send(szData,nDataLength);
        return 1;
    }
private:
    CAnswerToken m_Token[500];          //聚合类不面向资源，面向对象进行聚合
};

```

在笔者的工作中，经常观察到程序员会犯上述错误，这里提醒大家，当需求不断变更，原有设计的类已经变得太大太复杂的时候，要及时调整设计，不断将内部数据及其方法，单独封装子类，避免出现复杂度瓶颈，造成 bug 隐患。

3.5.2.5 习惯性接口

笔者在设计类的过程中，一般有些习惯性的标准接口函数，使用起来很便捷，这里建议大家使用。这些接口原则上都是公有方法，可以选择性使用。

1、ICanWork 接口，很多类都有正常工作的条件，如内部的缓冲区申请成功，socket 有意义等，如果每个内部函数都去检查每个调用条件，做防御性设计，显然程序不够简洁。笔者习惯性的用这个接口来解决。这里是一个书写实例：

```

bool CRelayAppServerKernel::ICanWork(void)
{
    if(!m_pGetRequestCallback) return false;    //每个判断失败都返回 false
    //...
    return true;                                //直到最后返回 true
}

```

这样，以后的所有接口函数，仅需要简单判断 if(!ICanWork() return;），即可做对象安全防御性设计。

2、Start、Stop 接口，商用数据传输工程中，有很多并行计算，笔者习惯于将为某个业务服务的线程，放到对应的类中使用静态函数完成，这在程序的易读性上有好处。但是，实际工程需求中，通常一个线程的起停和其对象的构建是异步的，因此，把启停线程的语句放到构造函数和析构函数中并不合适，笔者习惯于用 Start 和 Stop 两个接口单独来完成这个工作，保证异步的灵活性。

推而广之，一般说来，笔者习惯于把对象内部资源申请和释放，放在构造函数和析构函数，把运行期的变量初始化动作和恢复，放到 Start 和 Stop 中，这个设计，有点类似于 MFC 中 InitDialog 和 DestroyDialog 的设计，都是把运行期初始化动作和资源申请动作拆分，保持异步灵活性。这两个函数通常构型如下

```

void Start(void);
void Stop(void);

```

3、IAmWorking 接口，由于上述 Start 和 Stop 的存在，因此很多时候我们需要判断对象内部线程是否正确启动，因此，需要有个查询状态的接口，本接口即为此目的而设置。通常构型如下：

```

bool IAmWorking(void);

```

不过，近期笔者开发的习惯，这个函数有取消的趋势。因为笔者在后文总结的一个线程控制锁，有个接口可以查询到这个信息。这在后文有更详细的论述。

```

bool CthreadManager::ThreadContinue(void)
{
    return m_bThreadContinue.Get();
}

```

3.5.2.6 工具类

这算是笔者在实践中总结出来的一条经验，基本理念脱胎于模板的设计理念。先举个例子：

笔者在以前的一次工程中，遇到一个需求，我们需要用哈希型数据库来完成，对于数据的检索，我们需要使用树形目录，类似于/ROOT/ID/NAME 这类的目录来检索，这就遇到一个问题，我们在程序设计中，需要不断把“ROOT”、“ID”这类关键词，拼接成树形目录的表示方法供检索，或者不断把一个树形表示逆向拆分成每个单词来使用。

由于这类工作本身很简单，但工作量非常大，同时分散在程序的各个角落，这直接造成了程序员疲于奔命。bug 非常多。

笔者发现这个问题，思考了一下，建议一个程序员写了个 CPath 的类。其基本构型如下：

```
class CPath
{
public:
    //...
    void Tree2Normal(void); //树形逆解析为单词（每个单词表示一级树枝）
    void Normal2Tree(void); //普通单词拼接为树形表示
    char m_szTree[1024];    //树形表示
    char m_szWord[7][64];   //7 级目录的单词表示
};
```

这样，每次需要转换，程序员只要简单把原始数据拷贝到指定缓冲区，然后调用一次转换函数，即可从另外一个缓冲区获得对应的数据。所有字符串拷贝的检查，拼接和拆分的判断逻辑，都放到内部完成，外部程序无需理会。

这是一个典型的工具类设计，即所有方法和缓冲区都公开的类，完全就是为了封装一段算法，简化调用模型。实际用起来，这个效果很好，我们的核心数据库管理模块，用了这个方法后，代码迅速从 15000 行缩减到不到 2000 行，并且，所有的 bug 不见了。

工具类的写法，在笔者后文展示的库中还有很多，大家可以进一步熟悉，这类工具，一般没有一定之规，完全是程序员设计中，根据复杂度，需求，临时决定的封装，因此，笔者这里仅仅建议大家建立这种意识，灵活使用。

3.5.2.7 粘合类

这个应该算笔者自己起的一个名字了，因为笔者确实想不出，该怎么给这种类命名。看起来，它有点像 C++ 的模板的功能，但是，确实不是模板能够替代的。这应该算是工程实战中研究出来的一种写法。

1、C++ 和内存池的问题

前文我们说过，在 7*24 小时运行的服务器或嵌入式设备运行中，很多时候，内存碎片是个大问题。如果不加以解决，即使程序完全正确，在将来的某个时刻，也可能会因为内存碎片过多，导致大块内存申请总是失败，而引起服务的停止。

在后文的工程库中，大家可以看到内存池的一个设计实例，基本的思路非常简单，就是在系统堆管理器与应用程序之间，设计一个内存管理器，执行“只申请，不释放”原则。

对上层应用程序，内存池接收所有的内存申请请求，并查找内部有无合适大小的空闲块，如果有，则直接返回该内存块，而不必向系统堆申请，实现内存块重用，只有没有合适内存块的时候，内存池才会真正向系统堆申请内存。

同时，对所有的内存释放请求，也不是真实释放到系统堆，而是内部保留下来，以备下次重用。

这在系统堆看来，这段应用程序在运行期间，只申请内存，不释放内存，自然也就没有内存碎片了。而我们的服务程序，由于很多业务都是“短服务”，即快速响应一笔客户交易，在业务中申请的内存，完成服务后会立即释放，不会长期占用，因此，内存块重用度很高，自然，也不回因为这个“只申请，不释放”的原则，导致内存溢出。

这样做，当然能解决内存碎片问题，但是，我们知道，这种分配内存的方式，是基于纯 C 的 malloc 在工作，自然不可能完成 C++ 创建对象时，自动调用构造函数的功能，即它没有 C++ 语言中 new 这个语句的功能。

这说明，内存池只能服务于纯正的数据型内存块请求，如结构体，缓冲区之类的申请，不能服务于对象。

但是我们知道，在 C++ 程序开发中，大量使用动态对象，很多类，由于构造函数有参数需求，也根本无法以类成员变量形式自动创建。

这就带来一个问题，即使我们实现了内存池，有很大一部分动态内存申请工作，我们仍然无法管控动态对象创建那部分内存，实际上还是无法满足清除内存碎片的需求。

2、C++ 类和对象深入研究

笔者经过研究，发现在 C++ 中，类和结构体其实很相似，都是由变量区和函数指针表构成的。其中，函数指针表，包含所有内部成员函数的指针，指向的是编译器确定好的函数代码位置，每个对象均指向同一点，即任何一个类的每个对象的函数指针表，内容是相同的。

而每个对象拥有自己独立的私有变量空间，即一个类实例化的 a 和 b 两个对象，其变量空间不一样，因此，a 和 b 的成员变量可以各自保留各自的值，互相不会干涉。如下例：

```
class CClass
{
public:
    CClass() {}           //这些函数在对象中均为指针
    ~CClass() {}          //指向基栈中的函数代码
    Start() {}            //所有对象，这部分指针都一样
    Stop() {}
private:
    int m_nIndex;         //每个对象都拥有自己的私有变量区
    bool m_bICanWork;     //这些变量，根据创建方式的不同，处于浮动栈或堆中
    static int m_nIUse;   //这个是静态变量示例
};
int CClass::m_nIUse=0;    //这是 C++ 特性，全局初始化类静态成员变量的标准写法
                        //必须这么写，强制编译器在基栈创建该变量区域
                        //否则，编译时会由于 CClass::m_nIUse 符号未创建
                        //而发生 link 错误。
```

对象的私有变量空间，根据创建方式的不同而不同。当对象是静态实例化，即作为某个函数的内部私有变量被直接实例化时，其函数表和私有变量空间位于浮动栈。而当它是以 new 方式动态创建时，这些内容被创建到远堆上。

举个例子，针对这个 CClass，我们在程序中实例化两个对象。其中第一个，我们在函数中以静态方式实例化为 a，而另一个，我们利用 new 实例化为 b（指针为 pb）。

```
void Func(void)
{
    CClass a;             //这是在浮动栈，由系统自动创建的对象
    CClass* pb=new CClass(); //这是由我们的程序，使用 new 在远堆创建的对象
    //...
}
```


那么，我们来看看图 3.1，可以看到这些关联关系。



图 3.1: C++类对象内存分配

如图所示，对象 a 处于浮动栈，b 处于远堆，二者都有函数指针表，并且，所有的函数指针表均指向代码段相同区域，二者完全一致。但不同的是，a 和 b 各有其私有变量区，这些变量是完全没有关系的，这使得各个对象的成员变量不会互相干涉。

这里面特别值得一提的是 `m_nIUse` 这个变量，这在函数内被声明为一个静态成员变量，它的处理与其他变量不同。由于是静态的，它在编译期间，即由编译器定址，并放置在基栈里面。以后所有的对象，仿照函数指针区，针对该静态变量也仅仅是一个指针，指向的是同一个位于基栈中的变量。因此，静态函数是一个类的所有对象共享的，即 a 修改 `m_nIUse`，b 可以读到该变量的新值。这里笔者举个例子：

```
class CClass
{
public:
```



```

    CClass(){}
    ~CClass(){}
    int c1;           //c1 是普通成员变量
    static int c2;    //c2 是静态成员变量
};
int CClass::c2=0;     //通过初始化动作，强制编译器实例化静态变量，构建符号表，
                    //否则后续访问会发生 link 错误，找不到符号 CClass::c2
void Func(void)
{
    CClass a;         //这是浮动栈对象
    CClass* pb=new CClass(); //这是远堆对象
    if(!pb) return;   //防御性设计
    //开始赋值试验
    a.c1=1;
    a.c2=2;           //注意，这是静态变量 c2 第一次赋值，2
    pb->c1=3;
    pb->c2=4;          //注意，这是静态变量 c2 第一次赋值，4
    printf("a: c1=%d,c2=%d\n",a.c1,a.c2); //打印 a 的变量值
    printf("b: c1=%d,c2=%d\n",pb->c1,pb->c2); //打印 b 的变量值
    //程序退出，摧毁动态对象
    delete pb;
    pb=null;
}

```

最后运行结果：

```

a: c1=1,c2=4    //请注意，a、b 两个对象的静态成员变量 c2，均由于第二次赋值，变成 4
b: c1=3,c2=4    //而 a、b 对象的普通成员变量 c1，则各有其值，互不干涉

```

如本例所示，正是由于前文所述，类中静态成员变量是由编译器确定其运行期地址，并提前初始化在基栈中，所有的类实例化对象共享基栈中的变量存储空间，因此，a 和 b 两个对象中的 c2，以及使用 CClass::c2 表示的 c2，其实是一个变量，对象中，仅保留这个变量的指针，所有的指针，指向相同的地址区域。

静态成员函数与函数指针区的内容很类似，每个对象也仅仅保留一个指向代码的指针，所有对象的指针内容一致。此处不再赘述。

3、我们对 C++对象的结论

通过上文，我们可以得出结论，C++的对象的数据组织，主要包括以下三个部分：

- 1、函数指针区：包含所有成员函数指针，包括静态成员函数指针，各对象一样
- 2、静态成员变量区：包含所有静态成员变量指针，各对象一样
- 3、普通成员变量区：包含所有成员变量实例，每个对象各有其私有空间，互不干涉

那么，我们针对本小节的问题，是不是可以有一个初步的结论，C++所有的对象，唯一差异的，其实就是其普通成员变量的内容，其他都一致。其实每个对象最大的差异性，也就是这个私有变量空间。如果有了这个结论，笔者发现，前文所述内存池的用法，在 C++的对象模型下，就不是无解的问题了。

设想一下，如果我们不使用 C++编译器的特性，不允许它自行管理每个对象的私有变量空间，而由我们在内存池申请，每次实例化对象前，由我们申请好内存后传入，对象摧毁后，再由我们代为向内存池释放，则我们可以把一个类对象所有的私有数据空间，利用内存池管理起来，最大限度地避免了内存碎片的危害。

这里笔者举一个例子：

首先，我们假定一个类，需要处理一个 5k 的整数数组，里面有数组缓冲区和数组使用指针两个变量，这个类我们考虑要利用内存池技术，尽量减小内存碎片的危害。因此，我们需要先定义这个类的成员变量区。

```
//这是 CClass 类使用的数据区
typedef struct CClass_Data_Area_
{
    int m_nIndex;                //这是数组占用的索引
    int m_nDataArray[5*1024];    //这是一个整数数组，k 大小
}SCClassDataArea;
const ULONG SCClassDataAreaSize=    //习惯性写法，申请结构体马上
sizeof(SCClassDataArea);            //定义其尺寸，方便后续内存申请
```

这里，我们使用一个结构体来管理未来的类所有的普通成员变量，然后，我们的类写法和传统的 C++ 类就稍微有点不同了。

```
class CClass
{
public:
    CClass(SCClassDataArea* pMyDataArea)    //构造函数要求传入变量区
    {
        m_pMyDataArea=pMyDataArea;        //内部保留，以备使用
        if(m_pMyDataArea)
        {
            m_pMyDataArea->m_nIndex=0;    //此处做变量初始化
        }
    }
    ~CClass()
    {
        //注意，什么都不做，m_pMyDataArea 这个数据区不是自己申请的，
        //根据“谁申请，谁释放”原则，不该本对象释放。
    }
    bool ICanWork(void)                    //由于本类依赖外部传入的指针
    {                                      //因此，必须设置该公有接口
        if(!m_pMyDataArea) return false; //以查询是否可正确工作
        return true;
    }
    void DoSomething(void)                  //功能性公有方法示例
    {
        if(!ICanWork()) return;           //注意，所有公有方法需要先防御性查询
        //...
    }
private:
    SCClassDataArea* m_pMyDataArea;        //内部临时指针，指向本类数据区
                                           //数据区由外部调用者申请，并传入
};
```

大家可以注意，这个 CClass 类只有一个成员变量，就是参数结构体指针 **m_pMyDataArea**，其余所有业务相关变量，已经被整合到参数结构体中，并且，构造函数要求外部传入参数缓冲区，而不是内建。特别请注意的是，由于这个缓冲区不是本对象自行构建，因此，析构函数不要释放，由传入参数缓冲区的调用者释放。

那么，我们来看看函数调用实例：

```
void Func(void)
```

```

{
    SCClassDataArea* pObjData=                //此处使用内存池申请
        MemPool.malloc(SCClassDataAreaSize);
    if(!pObjData) goto Func_Obj_Data_Area_Destroy;
    {
        //此处是 pObjData 作用域开始
        CClass* pObj=new CClass(pObjData); //此处将内存区域代入到对象中使用。
        if(!pObj) goto Func_Obj_Destroy;
        {
            //此处是 obj 对象作用域开始
            //...
        }
        //此处是 obj 对象作用域结束
Func_Obj_Destroy:
        if(pObj)
        {
            delete pObj;
            pObj=null;
        }
    }
    //此处是 pObjData 作用域结束
Func_Obj_Data_Area_Destroy:
    if(pObjData)
    {
        MemPool.free(pObjData);                //此处使用内存池释放
        pObjData=null;
    }
    return;
}

```

如本例，由于这个类使用了特殊的设计方法，我们调用函数于是可以通过内存池来申请其缓冲区块，并传入到类对象中发挥作用，当所有的使用完成，我们的调用函数向内存池释放该内存块。由此，我们实现了类对象私有数据区的内存池管理，也就部分解决了动态对象内存碎片管控的问题。

当然，这里可能某些读者会指出，在 Func 示例函数中，pObj 这个对象还是 new 出来的，没有受到内存池管理，这种情况确实存在。不过，我们经过分析，可以发现如下特点：

1、CClass 这个类产生的对象，由于没有了内部高达 5k 的缓冲区内存开销，其实已经很小了，简单估算一下，其对象内部包含 4 个函数指针和一个私有变量区指针，我们知道，32 位操作系统下，指针都是 4Bytes，因此，该类产生的对象，最大长度为 20Bytes 左右，属于很小的内存块。

2、我们知道，内存碎片主要的成因，就是远堆由于多次的申请和分配，导致最后内存中有空区，但每个空区都很小，不足以应对大块的内存申请需求，如果我们按通常设计，CClass 内含 5k 的数据区，这基本上算大块了，则在内存碎片的申请中，很难通过，但如果只有 20Bytes 的小内存块，则几乎不存在这个问题，内存碎片影响不大。

我们知道，C++ 中的对象，通常是一堆数据与该数据相关方法的集合。但我们为了满足内存池管理要求，取消了对对象与数据的绑定，而仅仅融合了处理方法，这种写法，笔者称之为“粘合类”，即该类本身无数据，只有方法，完全是粘合到一块内存区去完成功能。

4、粘合类的实际使用方向

通常，粘合类主要用在服务器和嵌入式开发中，主要应对的就是如前所述的内存池管理需求。

不过，在笔者近期研究 Google App Engine 云计算平台时，有了一点新的看法，在云计算中，粘合类可望起到很大的作用。

我们知道，云计算的特点，是中心服务器集群提供统一的运行平台，允许客户自行在其上开发服务器应用，并为自己的客户端服务，这有几个特点：

1、云计算平台不是单独的 PC 机，仅仅是虚拟出来的一个任务执行机。

2、云计算通过服务器集群实现服务，针对同一服务，a 和 b 两个客户的两笔访问交易，可能是在同一台计算机上运行，也可能不是。

3、由于上述原因，云计算中的任务线程，没有共享的磁盘空间，没有共享的内存空间，a 交易和 b 交易，不能想当然认为可以互相通过全局变量或全局对象传递数据。

4、以 Google App Engine 云计算平台为例，数据的永久性存储只能通过数据库 api 完成，google 提供 mem cache 机制加速数据库中数据的访问，但不保证 mem cache 位于内存中的数据，能被所有的服务进程看到。

这就带来一个问题，作为一个网络服务器，大多数时候，我们针对每个客户的服务大多数时候是相对独立的，比如 a 和 b 用户分别修改自己的个人信息，这类交易一般都是互不干涉，独立完成。

但是，作为一个整体性的网络服务，总有一些业务是同时为所有客户服务的，如网站的总页面访问量计数器，此时，根据 GAE 的建议，计数器中存储的数值，只能放在数据库中作永久性存储。对于每笔交易的线程而言，很难通过一个对象来访问。

此时，笔者建议可以考虑使用“粘合类”，即构建一个纯提供访问方法的类，而对象的私有数据是贴合到数据库数据的一个内存映像上，当对象终结时，数据本身并不销毁，而是会被其他类似的服务进程，启动新对象继续使用。以此实现内存的高效访问。

当然，由于 Google App Engine 等云计算平台通常不提供 C++ 语言环境，上述粘合类可以考虑在 Python 或者 Java 等其他面向对象语言中实现。

3.5.2.8 “多次起停”与“不可重入”

很多 C 程序员在转到 C++ 下开发时，通常有个思维误区，即忽视“多次起停”与“不可重入”的设计。

在 C 语言开发时，程序员关注的程序单位是函数，而函数一旦被运行，系统会为其建立独立的浮动栈，专门为其开辟一块内存区，存储其变量。因此，C 程序运行时，哪怕在并行计算时，一个函数的多个运行实例各自拥有自己独立的变量空间，不会互相干扰。

但 C++ 加入了对象这个程序单位，对象天生具有可重用性，由于对象指针的可传递性，经常出现一个对象的方法、属性，会在多个函数模块中被调用，而所有的调用，由于访问的是同一个对象，因此，实际上访问的是同一个内存区域，不同的操作使用的数据，在对象内部相当于是全局的，互相干扰。在多线程环境下，这类调用如果不做好锁防护机制，将会是一场灾难。

并且，即使我们关注了锁防护，但通常也会有一些遗漏。传统的 C 程序员，较少关注程序单位的多次重入问题，一个函数，被同时调用多少次，都没有什么 bug，但一个对象，是否可以允许同时访问，就是一个大问题。

举个例子，假如一个对象内部假如包含线程，提供了 Start 和 Stop 方法来起停，那么，通常 Start 就是不可重入的，即已经 Start 后，不能重复调用 Start 多次启动。因此，一般说来包含 Start 和 Stop，有显式运行态的类对象，必须解决 Start 重入问题。也就是说，内部必须有详尽的防御机制，即使外部多次调用 Start，也能有效阻挡错误的调用，避免 bug。

还有个关键点也提醒大家关注，实现这个不可重入性本身的逻辑，其实也需要线程安全才行，因为重复调用 Start 的主题，很可能不是一个线程。在后文，笔者将会提供一个不可重入锁的实例，帮助大家解决这一难题。

另外，很多时候，可能由于程序员失误，Stop 后，没有将所有内部变量仿造构造函数一样，赋初值，又或者 Start 前，没有做赋初值动作，并且，在后续程序中，对内部变量有原始值依赖性，这就相当于一个函数内部，变量转义，并且没有初值重置，很可能导致 bug。

这样考虑，就必须解决类对象的多次重入问题，简单说，有如下要点应该关注：

1、显式把对象工作模型分为初始化态和启动态，初始化动作，资源申请动作，放到构造函数中处理，相关释放工作放到析构函数处理。专门定义 Start 和 Stop 做多次重入的起停工作。

2、Start 之前做所有相关变量的赋初值动作，这样，每次启动，确保一个干净的环境，不会受到上次启动的影响。以此实现多次重入的问题解决。

3、Start 做防御性设计，不符合条件启动失败，并且，Start 做好不可重入设计，避免重复启动造成 bug。

4、原则上，构造函数主要完成动态内存申请等变量合法性准备工作，具体的赋初值动作放在 Start 中，以便多次起停。

3.5.3 其他要点

除了基本的类和函数的写法，完成一个商业化开发，还有一些程序书写的细节需要关注，这里简要列举：

3.5.3.1 .h 的标准写法

1、在书写.h 头文件时，主要应关注避免头文件重入，即一个头文件的内容，由于多个.h 的互相嵌套 include，导致被同一编译进程多次编译，这往往会造成编译错误。

解决方法很简单，做一个简单的编译宏规避即可。每次书写.h 文件时，笔者习惯性的写下如下语句：

```
#ifndef _h_File_Name_ //如果没有定义一个编译宏，则编译下列语句
#define _h_File_Name_ //既然进入，则马上定义，避免下次再度被编译
//...
#endif // _h_File_Name_ //编译宏结束符
```

在 VC++ 中，很多时候使用的不是这个模式，而是在.h 文件中声明 “#pragma once” 来解决，不过笔者还是习惯使用前一种方法。

2、.h 文件中不要声明实例。很多时候，这会直接导致连接（link）错误。比如我们希望定义一个常量字符串如下：

const char* g_szWord="Word";

如果我们把这句话写在.h 文件中，由于大型工程，一个.h 文件通常会被多个.cpp 文件包含，那么，每次编译器走到这句，都会产生一个 g_szWord 的符号，并且为其在基栈中分配空间，构建实例，这样，当连接时，一旦连接器发现一处 g_szWord 的调用，开始引用时，会发现多个合法的内存区，无法确定哪个是有效的，于是报错。

正确的做法是将上述语句写在.cpp 中，而在.h 文件中，仅仅书写一句声明：

const char* g_szWord;

这样，编译器仅仅会在该.cpp 的 obj 文件中为该变量建立实例，.h 中仅仅为声明，大家均知道有此符号，具体符号表中，地址仅有一个，就不会出错了。

其他类型的变量，对象的实例化声明，一般都有此限制，建议大家养成习惯，**.h 中不要生成实例。**

顺便说一句，这也是笔者喜欢用宏来定义常量的原因，宏由于是编译预处理阶段，直接的文本替换，因此编译期实际上看不到这个实例化过程，因此，反而下面的写法，写在.h 中是正确的。笔者个人认为比较方便。

#define WORD "Word"

3、每个函数在.h 中声明为 **extern** 属性，这其实是 C 的习惯，一个函数一旦被声明为 **extern**，即被宣布为外部的，这样，当它被放到某个库中编译，以独立工程模块 **obj** 或 **lib** 形式，参与其他工程编译时，就不会出错。很多 C 程序员都知道这一点。

这里仅仅提醒大家一下，商用工程开发，对代码重用度要求很高，且有很多工程库的维护工作来做，养成这个习惯，所有全局函数在.h 的声明中，请尽量使用 **extern** 做修饰，这样，当某天我们把整个模块入库时，其.h 文件可以不必修改，直接使用。例如：

```
//Debug 与控制台相关
//安全的字符串拷贝函数
extern void SafeStrcpy(char *pD, char *pS,int nCount);
//安全的获得符合 Internet 规范的时间字符串
extern int SafeGetNowTime(char* szBuffer,int nLength);
//安全的变参打印函数
extern int SafePrintf(char* szBuf,int nMaxLength,char *szFormat, ...);
//默认打印的 buffer
extern int SafePrintfl(char* szBuf,char *szFormat, ...);
```

4、除非必要，尽量以纯 C 方式输出接口。这也是一个容易忽略之处，很多 C++程序员一旦进入 C++领域编程，脑中习惯性地会模糊 C 和 C++的区别，认为二者通用。

但，C 和 C++毕竟是两种不同的语言，尤其是在编译后的符号表格式上差别很大。如果一个库的书写者和调用者都是 C 和 C++语言，这没有任何连接问题，都是正确的。

但我们知道，很多时候，我们需要书写 **api**，供其他语言参考调用，很多脚本语言，如 Java、PHP、Python 等都支持调用 C 的库来实现一些更加强悍的功能。这类跨语言的编程环境，C 的通用性要远好于 C++，因此，在我们某天需要向其他语言的程序员提供一段功能支撑代码时，通常必须以 C 的方式输出功能，才能被正确调用。**api** 的设计一般也是基于纯 C 而言的。

因此，在.h 文件中，如果不是必须输出类等 C++特性的话，不妨加上一些下面的修饰，可能会带来意想不到的好处。

```
#if defined(__cplusplus) || defined(c_plusplus)
extern "C" {                                //纯 C 输出开始
#endif
//...    正常的声明段
#if defined(__cplusplus) || defined(c_plusplus)
}                                           //纯 C 输出结束
#endif /*defined(__cplusplus) || defined(c_plusplus)*/
```

5、除非必要，.h 不要包含其他.h。

这也是经验之谈，很多时候，程序员书写程序比较随心所欲。当一个模块依赖另外一个模块时，会习惯性地.h 中包含对应模板的.h 文件，这就造成了多重依赖。

这在一个大工程整理各个程序员代码时，会出现比较严重的依赖问题，整合很艰难。同时，在工程库的维护中，也严重影响程序模块的独立性，使库整理非常困难。

比如这样一个例子，我们在 **a.h** 和 **a.cpp** 中定义了一个类 **A**

```
class A
{
};
```

然后，我们在 **B** 这个类模块中调用了 **A**，自然，我们需要包含 **a.h**，如下：

```
#include "a.h"
class B
{
```

```
A a;
};
```

自然，当以后类 C 需要调用 B 时，也就必须把 a.h 包含进去，实际上，也许 C 这个模块根本和 A 没有关系，不需要了解 A 的细节。但没办法，如果哪天我们需要提供 C 模块，则必须把 A 和 B 一并提供，否则编译失败。

我们知道，计算机编程语言的发展，实际上就是一个数据私有化的过程，强调减少全局信息，各个模块尽量不要知道不想管的东西，也尽量不要被别人知道私有的信息。这类牵连依赖关系，很可能将我们前期辛辛苦苦，通过模块设计构造的私有化模型破坏掉，是没有关系的两个单元互相看见对方，造成无谓的牵连，破坏低耦合的设计原则，造成无谓的 bug。

建议的写法如下：

在 b.cpp 中来包含 a.h，即把#include "a.h"这句话写在 b.cpp 中。

而在 B 的类声明前手动声明 A 即可。

```
class A;    //手工声明 A 是一个类
class B
{
    A a;
};
```

3.5.3.2 .cpp 的标准写法

.cpp 文件其实就是 C 和 C++ 语言开发的程序实体了，几乎所有的程序逻辑模块都书写在这类文件之中。各位程序员也比较熟悉，这里仅提示几个要点：

1、VC 的怪癖

微软公司在设计 VC 的时候，一定没有完全遵守 C 和 C++ 的标准，很多时候 VC 这个编译器有一些怪癖，比较突出的就是 stdafx.h 这个预处理文件。

我们只要任意开辟一个 VC 的工程，在每个.cpp 文件的第一句就能看见这样一句话：

```
#include "stdafx.h"
```

很多时候，这句话一点用都没有，但必须这么写上，哪怕是程序员自己书写的.cpp，也必须加上这一句。这在 gcc 等编译器看来，完全无可理喻。并且，据笔者观察，这个写法要求还很多：

- 1、必须写在每个.cpp 文件不是注释的第一句，否则出错。
- 2、必须确保在该.cpp 目录下有这个文件，该 include 不能带目录，否则出错
- 3、文件名大小写不论，但一定要叫 stdafx.h

不过，由于 VC 几乎已经是最普及的 C++ 编译器，因此，哪怕我们开发跨平台的工程项目，至少也需要 VC 的支持，因此，这点笔者没有任何办法，只有尊重 VC 的这一怪癖。

不过，要支持也不简单，首先，要从一个标准的 VC 工程中，拷贝一个 stdafx.h 的原文出来，放到我们的模块目录下。这个文件通常保持如下内容：

```
#pragma once
#ifdef WIN32
    #include <iostream>
    #include <tchar.h>
#else // not WIN32
#endif
```

其次，Unix 和 Linux 下对文件名大小写很敏感，一般建议不要用大小写混用的文件名，因此，需要手动把这个文件改为纯小写文件名，这样，Windows 下合 Linux 下均不会出错。

保持自己模块下每个子目录中放一个，保持自己每个.cpp 文件第一句都是这个包含语句，那么，程序方可在 VC 和 gcc 下均顺畅编译。

2、gcc 的怪癖

刚刚说完 VC，我们又来说说 gcc。其实 gcc 本身并没有什么特变古怪的癖好，大多是遵循 C 和 C++ 的标准来，不过有时候由于太遵守标准，也会导致不方便。

gcc 规定每个 cpp 的最后一行一定是空行，否则有一个编译警告出来，由于我们工程编译要求 0 warning，因此，这也必须关注，大家在书写 cpp 文件的时候，请注意最后一定要留一个空行了。

这样久了，笔者在书写一个新的.cpp 时，就养成了一些习惯，上来就先写如下几句话，基本能保证所有环境编译无误。

```
#include "stdafx.h" //VC 的怪癖，要尊重
#include "my.h"      //这是本模块自己的.h 文件
//正文...
//////////////////////END    //下面这个空行必须存在，gcc 的习惯要尊重
```

3.5.3.3 Makefile 的简单写法

99 年，笔者第一次在 DOS 下使用 gcc 的时候，并不是很看重 Makefile 的书写，因为那时的工程一般都比较短小，没有必要选择编译，简单做个 bat 的批处理就够了。再后来，笔者进入 VC 开发领域，由于都是 IDE 代为管理编译序列，也一直不太重视 Makefile 的书写。这也算是在微软的平台下开发久了，没有养成好习惯吧。

不过，由于近年来在 Linux 开发任务的增加，出于大型工程编译的需求，笔者不得不研究一下 Makefile，然后出了这么一个例子。

笔者第一次感到需要一个 Makefile 后，由于不会，就请求另外一个部门，负责嵌入式研发的同事帮忙写一个简单的 Makefile，然后笔者等了两天，那个同事没有写出来。笔者就着急了。这到底有多复杂，那么难写？

笔者走过去和他沟通，看到那位同志正在疯狂 Debug，为 Makefile 做 Debug。然后笔者检查了他写的 Makefile，感觉像天书。按他的说法，笔者的工程库目录太复杂，嵌套和递归关系也太复杂，导致写出来 bug 很多，无法编译。

Makefile 是最喜欢用编译宏的，而且，很多书上的建议是 Makefile 最好使用循环嵌套关系，即仿造程序的结构，构建多个模块的 Makefile，然后整合一个大的，据说这样在迁移模块到各个工程时非常方便。

但这直接造成了很多递归包含关系，整个 Makefile 我评价，其复杂度不亚于我们自己开发的工程，相当于用另外一种程序设计语言写了个程序。但这，只有一个问题，有必要吗？笔者做了如下思考：

1、Makefile 的书写，仅仅是程序编译期的一个过程，随时可能修改，应该尽量简化。避免太复杂逻辑，修改不便。

2、笔者对于太复杂的逻辑，特别是递归、嵌套一类的逻辑有种天生的厌恶，习惯简单化处理问题，Makefile 作为一个辅助工具，完全没必要占用太多程序员的时间，做这么复杂的设计。

3、从整个工程任务看来，Makefile 的工作量，应该不占用，或者占用很少的工时才对，像这样一个文件写两天，这在项目上，已经算一个中型模块的工作量了，成本上不划算。

4、Makefile 所有的这些设计，据笔者理解，仅仅是为了工程 Move 的时候少敲几下键盘，很多时候甚至还不少敲，对于这类设计，笔者一般排斥，笔者认为程序员不要怕打字，

很多东西，手工打一遍最好，一来过一遍脑子，少 bug，二来把中间过程直接暴露出来，对于后期查找 bug 有利。

基于上述理念，笔者请哪位同事别忙了，自己看了一些 Makefile 的书，写下了如下的标准段落，以后的 Makefile 照此办理，再也没有出错。

```
wb_relayer: \                                //定义最终的可执行文件名
    relayer.o \
    osdefine.o \                             //所有.o文件的列举，供连接使用
    ... ..
    tony_relayer.o                          //注意最后一个没有"\ "
    gcc -Wall -I. -o $@ $^ -lpthread -lstdc++ //标准编译命令

relayer.o:                                //分文件编译，产生.o
    gcc -c $<

osdefine.o:                                //手工书写路径名
    gcc -c $<

... ..

clean:                                     //起一个简单的 clean 命令，供以后使用
    rm *.o -rf
    rm wb_relayer -rf
```

这样的写法，看似很笨，所有的文件名，路径名，笔者必须手动输入，但由于编译过程为典型的单线过程，每一步都是程序员书写的，因此，反而错误很少。整个逻辑功能简单，仅仅定义一个编译，以及 clean，满足大多数使用场合。

因此，这里再次提醒大家，简简单单写程序，包括 Makefile。

3.6 基本语句的约定

笔者使用 C 语言快 20 年，C++ 语言也有十几年，应该说，对这两门语言，笔者有了较深的感情。但笔者还是不得不承认，C 和 C++，确实不是一门严谨的语言。原因很简单，其设计中，同义的内容太多，很多逻辑，可以使用多种方式实现。

这直接造成了很多人虽然都在用 C 或 C++ 语言写程序，但实际上用的是他自己的方言，在商用工程项目开发中，这直接带来的就是各个程序员自写一套，各自为战，同样一个循环，一个项目组可以出现好几种写法，虽然大家都是同一种语言，同一个平台，但由于书写习惯的差异，很多程序员其实看不懂彼此的代码。代码没有可重用性，严重影响项目团队的效率。

笔者在带领团队工作的过程中，为此非常伤脑筋，后来召集团队成员讨论，大家做出一些约定，对于很多非性能优化相关，非关键动作的常规写法，按照一定的标准和格式来写，这就是基本语句约定。

此处笔者强调一句，下述基本语句约定，仅仅是笔者长期工作的经验积累，以及一些过去的项目习惯，本书在此，仅仅是提供建议，并不是绝对要这么做才符合无错化程序设计规范，各位读者完全可以根据实际情况，直接套用，或者是自行约定。

3.6.1 判断语句，常量永远在左边

这其实是一个老话题，C 语言有个天生的缺陷，就是使用 “==” 来判断两个变量是否相等，这直接造成了很多时候，程序员由于少写一个 “=” 而导致的 bug。

举个例子：

```
void Func(void)
```

```
{
    int i=1;
    if(i == 0) ... //这里判断 i 是不是=0
}
```

但很多程序员往往由于笔误，写成如下形式

```
if(i=0) ...
```

此时，括号内为表达式，给 i 赋 0 值，同时，表达式返回 0，也就是该判断语句恒假，其后的语句永远无法执行。

这个错误曾经给笔者自己都带来很大困扰，因为程序员是人，笔者也是人，是人就一定可能犯错误，谁都不敢保证自己永远不写错。最要命的是，这个错误在编译器看来，本来就是正确的，语法上讲得通，合理又合法。

在笔者看来，“**看得到的 bug，都不称其为 bug**”，因为程序员只要看见 bug 了，总有办法修订，最怕的是看不见的 bug，无错化程序设计思想的核心，并不是绝对没有 bug，而是通过一系列方法，来让 bug 第一时间暴露出来，最好就是编译器报错，强迫程序员修改，那这类错误，一般都不再是 bug。

后来，笔者在微软公司的一本书上看到，其实这个问题很简单，只需要简单把常量写左边即可，如下错误：

if(0=i) ...

我们知道，常量是不可以被赋值的，因此，这个 bug 会在第一次编译时，被编译器检查出来，并报错，提醒程序员修改。

笔者在程序生涯中，强调一个观念：“**用一个好习惯，解决整整一个方面的 bug**”，既然这个问题有解决方案，那就很简单了，养成一个习惯，以后写判断语句，一律常量写左边即可。

这在大多数时候有用，不过，个别时候，如果判断的双方，都是变量，怎么办？笔者一般区别对待，如果参与比较的一方是传进来的参数，则通过 const 解决，如下例：

```
void Func(const int nIndex)
{
    int i=0;
    if(nIndex == i) ...    //此时，nIndex 被定义为 const 属性，可以看做常量
}
```

当然，在某些动态比较场合，确实比较双方都是变量，没有办法套用这个规则，则只有请各位程序员仔细一些了。不过，由于笔者已经养成了好习惯，每次遇到判断语句，习惯性地都要想一下哪个是常量，这种习惯也迫使笔者对判断语句具有高度的敏感性，因此，只要谨慎一些，一般也不会再出现这类 bug 了。

3.6.2 for(i=0;i<n;i++)

这是个很奇怪的话题，笔者提出这一写法，并在团队会议上强调有次数限制的循环，一律使用这种写法时，很多程序员都不理解。因为条件循环的写法太多了。for、while、do...while 等等。但是笔者坚持大家这么做。

原因是这样的，C 语言，其实计算机所有的语言，在定义数组的时候，一般都是从 0 开始计数，即 int a[10]，数组单元的编号是 0~9，这和人类的习惯很不一样，我们人类一般习惯从 1 开始计数，即 1~10。

大家不要小看这 1 的差距，很多循环次数计算错误的 bug，就是从这里开始的，通常情况下，我们遍历一个数组，用得最多的就是循环。如下面这个例子：

```
#define ARRAY_MAX 10
void Func(const int nIndex)
```

```
{
    int nArray[ARRAY_MAX];
    int i=0;
    for(i=0;i<ARRAY_MAX;i++)    //注意，此时 i 的取值是 0~9，符合 C 的数组编址习惯
        nArray[i]=0;
}
```

凡是数组操作，这几乎是唯一正确的写法。因为 i 的取值范围，正好和 C 语言数组编址习惯对应。

如果我们写成 for(i=1;i<=ARRAY_MAX;i++)，大家注意，此时 i 的取值是 1~10，那么，我们在使用数组时，被迫做地址修订，nArray[i-1]=0;，而很多时候，程序员可能会忘了做这个修订，于是，一次内存写出界就出现了，程序可能会随机崩溃。

在笔者过去审查代码中，还看见过一个更加奇怪的例子：

```
void Func(const int nIndex)
{
    int nArray[6];        //常量没有定名，6 很奇怪
    int i=0;
    for(i=0;i<=6;i++)
    {
        if(i==6) break;    //中途跳出不对
                           //最关键的，大家看这么写奇怪不？

        nArray[i]=...;
    }
}
```

当时一个程序员程序一直有 bug，笔者帮他修改，看到上述的段落，大家看，由于程序员没有养成良好习惯，循环语句中随手写出 i==6 的判断，出现问题了，循环次数多了一次，内存写错误，但他又没有细想这个问题，随手又写出了一句 if(i==6) break;来解决，程序累赘，bug 几乎不可避免。

最后笔者没有办法，找了个数组题目给他，要求他必须写成 for(i=0;i<n;i++)的形式，他终于明白了其中含义，以后再也没有出现类似的 bug。

这个规定其实很简单，叙述一下，**for 循环，一定从 0 开始，中间的判断一定是“<”，而不能用“<=”，一定是以 1 为步距单步递增，不能递减。**

这个习惯的反对者很多，举出的例子也多种多样，笔者也无法一一解答，这里仅仅列举几个常见的反对意见，供各位读者参考：

1、有朋友提出，有的程序需要递减逻辑，笔者回答，请把递减逻辑，改写成递增逻辑，中间用减号，不难的，当一个项目中都是递增逻辑时，大家的代码看起来会容易懂一些。

2、有朋友强调，很多计算，不是从 0 开始，i 值不符合计算约定，i 如果是浮点数怎么办？笔者回答，请参考前面严禁变量转义，循环控制变量，原则上不应该参与计算，请另行设置循环计算变量。而循环控制变量，一定是整型，总不可能有循环 0.1 次的说法吧。

3、有朋友强调，很多算法，这么写不出来，这笔者没有什么好说的，至少笔者写 C 和 C++ 程序近二十年，还真没遇到这么写写不出来的代码，好好动动脑筋，都可以改写成这种写法的。

4、还有的朋友强调，这样没有办法实现死循环，笔者回答，死循环请参考下面的 while(1)，不要用 for 实现，这仅仅是习惯问题。

3.6.3 while(1)

同上，由于 C 和 C++ 语言的多样性，死循环的写法也多种多样，很多开源代码中，可以看到 `for(;;)` 这种写法，都对，但是，笔者这里强调团队统一。而且还有个视觉醒目的作用。

`for` 已经被用做有限次数的循环控制，那么，如果无限死循环也用 `for`，会让很多人看昏头。因此，笔者给团队的建议，死循环，不能预知循环次数的循环，主要进行条件等待的循环，请一律使用 `while(1)`。至于为什么不用 `while(true)`，仅仅是为了简便。

这样，笔者和同事，只要在代码中看见 `for`，就知道这是有限次数循环，看见 `while`，就是死循环，非常分明。

3.6.4 不要使用 do...while()

笔者有时候，觉得 C 语言太慷慨了，仅仅是一个循环，就定义了好几种语句，`do...while()` 就是一种。

不过就笔者个人的习惯，一般回避使用这个语句结构，原因很简单，上述两个写法已经够用了，没有必要再用新的花招和技巧，使程序更加复杂，难以阅读。仅一个循环就搞多种写法，也不符合简单化程序设计的原则，很多时候，除了显示程序员很酷之外，其实并没有太多用处。

还有一个更加重要的理由，`do...while()` 是先执行，后判断，不管条件是否满足，先把循环体执行了一遍，这在某些商用工程中，可能出现防御漏洞，而且不好查找。原则上，**计算机做任何一件事，都应该满足一定条件，无条件地做事，实际上是不存在的**，而 `do...while()` 的存在，有点破坏这个原则。

比如这个例子，就是典型的逻辑歧义：

```
void Func(const int nIndex)
{
    int i=0;
    do
    {
        //...
    } while (0!=i);    //请注意，判断条件是 i 不等于 0，但循环体已经被执行一遍了
}
```

后来的程序员在读代码的时候，会非常混乱，书写者究竟想干什么？这会带来很多逻辑上的错误，但大家注意，程序本身是正确的，编译器不会报错，这个 **bug** 将必须由程序员在通读程序的前提下，肉眼查找。

笔者写了这么多年代码，很多次遇到 `do...while()` 的时候，都仔细地将其修改为直接使用 `while` 语句，从来没有出现逻辑错误。笔者也确实没有遇到过必须使用 `do...while()`，否则写不出来的情况，因此，建议各位读者，以后考虑回避使用这个结构。

3.6.5 i++和++i 问题

这和上一问题类似，其实都是多元动作的先后顺序问题，第一个是先操作后累加，第二个是先累加后操作。这个用法涉及很多具体的业务算法，很难讲该用哪个，不该用哪个。

不过，笔者从程序简化设计来说，建议“**从一而终**”，即选择一个方式，以后就尽量只使用这个方式，不要变，以免引起后续的阅读理解困难。

笔者通常只使用 `i++`，这么多年，也确实没有遇到必须使用 `++i` 的情况，慢慢就养成习惯了，这样，笔者在书写代码是，总是先操作后累加，程序原则一以贯之，极少 **bug**。

与之对应的还有 `i-` 和 `-i` 等问题，其实大同小异，不过，笔者由于前面一直习惯使用递增序，因此，`i-` 基本上不用。

从另外一个角度，即严谨变量转义的角度，其实这类写法都有潜在的转义隐患，前面我们讲了，笔者建议 `for` 循环永远递增序，因此，循环变量的累加，一般都是仅使用 `i++` 即可，其他的计算方式，都可能被用作变量转义，即一个变量 `i`，既是某个控制变量，又是计算的参与者，这很危险，也应该尽量避免。

3.6.6 请不要使用 “`?(...):(...)`” 结构

其实单就语法上讲，笔者并不喜欢 C 语言，就是因为 C 语言有很多奇怪的语句设置，笔者也很不理解。这个 “`?(...):(...)`” 结构，就很奇怪。各种语言，都有详细的判断分支语句，可以应对任何判断情况，C 的这个设计，有点莫名其妙。

笔者后来查证了半天，这个语句的设置，估计是 Unix 系统的时代，没有显示人机交互都是电传打字机，打印纸很贵的，那个时候，能少打一行是一行，才会有这个设计出来，这应该算作 Unix 程序员的怪癖，惜墨如金，比如好好的 `catalog` 命令，本意是列目录，Unix 硬给改成 `cat`，猫，严重误导消费者。

这完全不符合 C 和 C++ 无错化程序设计方法中，关于程序简单易懂的原则，一来很重复，二来没有必要，三来严重影响程序的可读性，因此建议不用。笔者从学习 C 语言的第一天开始，就没有用过这个结构（所以这里连例子都举不出来），都是老老实实在地写成 `if...else...` 判断语句，也没发现有什么程序写不出来。

3.6.7 善用大括号 `{ }` 缩小作用域

大家都知道，在 C 和 C++ 语言中，大括号 `{ }` 是很重要的元素，一个函数，一段逻辑，一个类，一个结构体，都要用大括号来包容，以体现其逻辑独立性。不过，笔者在实战中发现，很多时候，大括号还有其他用途，用好了，可以有效提升程序的品质，降低 bug 率。

3.6.7.1 限定变量作用域

很多读者都知道，C 和 C++ 有很大的区别，最主要的就是 C 是面向过程的（OP），而 C++ 是面向对象的（OO），笔者这一代程序员，一般都是从汇编、C 这条路上走过来的，因此先接触的是 OP 思想。在学习 C++ 的过程中，笔者和很多人一样，陷入思维混乱，很难理解 OO 的对象思想。

这种情况一直持续到笔者读到一句话：“**对象就是一些数据，以及针对这些数据所有方法的集合**”，这让笔者联想到当年第一次接触结构化程序设计时，理解的“高内聚，低耦合”的开发思想，这才把 OO 的概念理解通。在笔者看来，C++ 的面向对象的程序设计思维，就是结构化程序设计的一次编译器强化和具体体现而已。

而结构化程序设计思想，核心就是强调数据私有化，减少甚至灭绝全局变量，避免数据的无规则滥用，以便在大型工程开发中，降低 bug 率，这和 C++ 的类中，明确以 `private` 强调数据私有的设计思想何其类似。以这个思想思想，笔者再来看开发语言的发展，其实就是一个数据私有化的发展历程。

举个例子，最早的汇编语言，其实是没有私有数据的，所有的数据都是全局编址，全局可见，自然也就没有数据安全性可言。任何一个程序员，只要添加一段代码，他甚至都不必理解某个内存地址的原有含义，就可以直接修改，这给工程化协同开发，带来巨大的代码隐患。

C 语言也有类似的缺陷，编译器上允许全局变量的存在，导致很多 C 代码变量满天飞，bug 一大堆。到 C++，由于引导正确，虽然还有全局变量的设置，但笔者发现程序员用得就比较少了。

笔者这里说这么多，其实是想阐述一个道理：一段程序，变量私有化程度越高，程序的 bug 越少。换言之，也可以这么说：一段程序，每个变量的作用域越小，程序的 bug 越少！

基于这个理念，笔者的习惯，即使在一个函数内部，笔者有时候也喜欢使用大括号{}来显式确定变量的作用域，即一个函数内部，笔者也界定某个变量为局部变量，效果非常好。比如下面这个例子，减小了 i 的作用域，提升了程序的安全性。

例 3.6.7.1.1

```
void CMyClass::Func(void)
{
    if(!ICanWork())
        goto CmyClass_Func_End_Process;
    {
        int i=0;          //此时，i 的有效作用域仅限于这两个大括号之间
        //...             //注意，如果没有大括号，gcc 编译会出错
    }
    //...
CmyClass_Func_End_Process:
    return;
}
```

这个应用方法在某些时候，可以用来规避变量转义问题。

例 3.6.7.2.2

```
void CMyClass::Func(int nCmd)
{
    switch(nCmd)
    {
    case 0:
        {
            //使用大括号，限定内部变量的作用域，相当于 inline 函数
            int i=0;    //请注意，这两个 i 不是同一个变量，不算变量转义
            //...
        }
        break;
    case 1:
        {
            int i=0;    //请注意，这两个 i 不是同一个变量，不算变量转义
            //...
        }
        break;
    default:
        break;
    }
}
```

请注意，在例 3.6.7.1.1 中，还体现了大括号的另外一个好处，C++语言有个限制，即不允许在 goto 语句之后声明变量，因为编译器担心如果跳转写得不好，可能会导致某个变量未声明而使用，VC++对此不会报错，但 gcc 会报错。

但在C和C++中,函数内部大括号的真实含义是开启一个新的浮动栈,很类似一个 inline 函数被编译器展开到这里的情况,很多的时候,出于防御性设计的需要,我们在一个函数开始,通常都有大量的参数合法性判断,再加上“函数只有一个出口”原则,我们需要 goto 到函数最后。

这就会带来一个问题,即函数程序逻辑自身的临时变量在哪声明的问题。原则上,声明点离使用处越近越好,因为直观,但由于上述原因,将会被迫在函数开始声明,函数阅读会有一定困难。

这时,我们可以仿造例 3.6.7.1.1 的用法,在防御性判断之后,使用新的一重大括号{} 开始正式逻辑,此时即可自由声明变量。保证程序的紧凑性。

3.6.7.2 突出显示二元动作的内部逻辑

笔者还有个习惯,在很多二元性动作,特别是加锁和解锁逻辑时,将中间的逻辑段落故意用大括号包住,并且缩进,这在浏览程序时一目了然,非常便于理解和查错。如下面的例子:

```
void CMyClass::Func(void)
{
    //...
    m_Lock.Lock();
    {
        //...          //中间的逻辑段落一目了然,非常便于直观了解锁的作用域
    }
    m_Lock.Unlock();
    //...
}
```

3.6.7.3 函数型宏定义预防歧义

在很多书写较为复杂的函数型宏时,为了让宏内的代码段和未来展开处的代码块,从格式上显式区分,避免编译器误读,也需要用到大括号。如下面的例子:

```
#define TONY_DEL_OBJ(p) if(p){TONY_UNREGISTER(p);delete p;p=null;}
```

这是笔者常用的删除对象的函数宏,大家可以看到前后都有大括号包围。这是因为编译器在展开宏的时候,完全是按文本来展开,并不理解展开代码的含义,如果没有一个显式的分隔符,有时候会和后续代码粘连,产生歧义,大括号可以作为有效的分隔符使用。

3.7 请使用 goto 语句

这本来应该是个小话题,可以放到前一节的基本语句约定中,但笔者思考了一下,决定还是单独开辟一个章节来说明这个问题。原因很简单,这个话题太敏感了,几乎每一个笔者提出这个概念,不管是同事,还是网友,都遭遇到强烈的反对意见。

笔者自己也知道,从 1988 年,笔者第一次接触到结构化程序设计和模块设计理念,学到的第一个概念就是“**不要使用 goto**”。笔者自己差不多也有十年左右的时间,回避使用 goto,但近年来,笔者反而改回来了,越来越喜欢使用,并将其定为自己项目团队的开发规范,这是为什么呢?

我们知道,C 语言从 1969 年诞生,到现在差不多 40 年了, goto 语句从一开始就被设计为 C 语言的基本语句,出于维护标准起见,没有取消,这可以理解,但笔者很好奇的是,C++语言,这门绝对结构化的面相对象语言,居然也保留了 goto,甚至很多其他语言,在不断面相对象的进化中,也保留了 goto,(比如 C#),这是为什么呢?

直到笔者看了一个微软公司的 PPT，里面提到程序鲁棒性的十大原则，笔者才恍然大悟：万事要一分为二看待，goto 并不是绝对不好，如果善用 goto，能简化程序设计，提升程序的鲁棒性。

3.7.1 函数只有一个出口的原则需要 goto

这个前文 5.3.3 节已经说过，函数只能有一个出口，不允许中间 return，或者 break、continue，在其中的示例里，笔者也展示了 goto 的用法，这里不再赘述，大家可以参考前文的例子。

3.7.2 谁分配、谁释放的原则需要 goto

这是微软的例子，笔者深以为然，这里摘录如下：

例子一：不用 goto，想想需要申请的指针是 10 个的话，程序怎么写？

```
void Func(void)
{
    char* p1=null;
    char* p2=null;
    char* p3=null;
    p1=(char*)malloc(10);    //注意：此处 p1 已经分配，后文必须释放
    if(!p1) return;
    p2=(char*)malloc(10);    //注意：此处 p2 已经分配，后文必须释放
    if(!p2)
    {
        free(p1);
        p1=null;            //由于 p2 的分配失败，必须释放 p1
        return;
    }
    p3=(char*)malloc(10);    //注意：此处 p3 已经分配，后文必须释放
    if(!p3)
    {
        free(p1);           //由于 p3 的分配失败，必须释放 p1 和 p2
        p1=null;            //这种累加式增长会随着使用的指针的增多，而大量增加
        free(p2);           //如果是 10 个指针，程序会变得很长
        p2=null;
        return;
    }
    //很不幸，很多构造函数都负责预分配本类所有的指针，上述逻辑，在构造函数中很常见
    //不止是 malloc 和 free，new 和 delete，加锁解锁等所有二元动作，都有类似问题
    //...
    if(p1)                  //虽然上面已经写了很多释放代码，但函数结束时仍然需要书写
    {                        //重复的代码输出，导致程序非常累赘
        free(p1);
        p1=null;
    }
    if(p2)
    {
        free(p2);
        p2=null;
    }
}
```



```

    if (p3)
    {
        free(p3);
        p3=null;
    }
}

```

例子二：用 goto

```

void Func(void)
{
    char* p1=null;           //此处，赋初值非常重要
    char* p2=null;
    char* p3=null;
    p1=(char*)malloc(10);
    if(!p1) goto Func_End_Process;    //简洁的 goto，将所有释放动作归结到一点
    p2=(char*)malloc(10);
    if(!p2) goto Func_End_Process;
    p3=(char*)malloc(10);
    if(!p3) goto Func_End_Process;
    //...
Func_End_Process:
    if (p1)                  //初值判断出哪些需要释放，然后保证安全释放。
    {
        free(p1);
        p1=null;
    }
    if (p2)
    {
        free(p2);
        p2=null;
    }
    if (p3)
    {
        free(p3);
        p3=null;
    }
}

```

从上述例子中可以看出，二元动作需要后续在现场恢复，但如果一个函数中处理多个二元动作，不用 goto 的话，则会导致大量重复但是必要的代码，这一来使程序不够简洁，二来也增加 bug 的机会。

3.7.3 商用工程要求 goto

这不是危言耸听，笔者分析了结构化程序设计思想，发现其核心思想就是模块化设计，每个模块具有高度的内敛性，所有的运行期错误，原则上内部处理完成，不要麻烦调用者来处理。

在商用工程中，尤其如此，很多程序员的模块，并不是自己调用，而是其他伙伴调用，有的伙伴甚至在外地，国际化合作的结果，甚至不是本国人。这给沟通带来巨大的不便，

如果一个模块不能高度内敛，具有太多的调用限制，返回了太多的错误类型，则会造成调用者大量的不便，甚至造成人为的 bug。

另外，商用工程，一般具有割裂性，和你通信的另一方，可能根本不是程序员自己开发的，甚至根本不是本公司的产品，（各位读者可以想象一下 IE 浏览器的通信对象有多少种，调用 IE 内核的应用程序又有多少种）。模块的开发者，不能指望自己的调用者来帮助自己做异常恢复。

这说明，任何一个商用工程的模块，其实都应该是成熟的商业产品，自动纠错，自动异常恢复，不管是正确还是错误的调用，首先保证自己有足够的保护机制，不能死机，错误的调用，仅仅是返回一个失败值，宣告服务失败而已。

所有这些，说明一个合理的商用工程模块，应该具有产品级的安全性，而这体现到程序设计上，就是大量的防御型设计和异常处理，大量的分支判断语句。

对此，包括 C++ 等很多面向对象的程序设计语言给出的建议，是使用异常，但笔者认为，异常有个天生的缺陷，就是对调用者的依赖性。根据 C++ 的设计原则，被调用者在检测到某种异常状况后，一般是抛出一个异常，然后逻辑终止。这个异常，如果在本逻辑段内没有被 cache 捕获，则会依次上传给每一级调用者处理，如果大家都不处理，则最终会由操作系统捕获，终止程序报错。

这说明什么，只要有一个哪怕很小的异常，由于程序员忘了、笔误，或者某种沟通不到位，没有被刻意去捕获，则会导致一次程序崩溃，可能这么设计是出于好心，但就本质看，一个 7*24 小时服务的服务器程序，崩溃了，服务终止，这已经是最大的故障了。

因此，在商用工程中，乱抛异常，和直接调用 exit 差不多，这样的异常设计，很多时候还不如不设计。一次小规模的内存写出界，可能仅仅是某笔客户交易数据被写坏，单笔交易失败，一次 double lock，也可能仅仅使某个线程挂死，导致系统服务资源变少，至少服务不会终止，但没有控制地乱抛异常，则完全可能导致程序直接崩溃，服务中断。

还有个最麻烦的问题，运营期的异常本来就是由于各种各样复杂的网络环境造成的，与异常时的网络状况密切相关，这一来在实验室中无法模拟，测试没有办法检验所有的异常状况，程序员不能保证杜绝，二来，出现运营期异常，一般也无法跟踪分析，通过修改代码来适应，程序员几乎无解。这是非常可怕的现象，但又确实存在。

并且，商用工程项目，往往会运营在某个电信商的机房中，很多还是全球运营，一次崩溃后的重启，单机用户可能仅仅是简单按一下 Reset，而我们的客户，则可能付出高昂的维护成本。

比如说，半夜起来打的去电信机房，按一下 Reset。

上述事例说明，商用工程具有高度的结构化编程需求，要求每个模块具有高度的内敛性，必须内部处理所有异常状况，不能对任何其他模块有依赖关系。因此，笔者建议，这类开发中，应尽量减少异常机制的使用，避免人为的缺陷。

那么，我们商用工程程序员，在模块内部的异常处理中，可能就只有使用 goto 语句一条路来解决问题了。每个函数仅有一个出口，任何异常都放弃本笔交易，跳转到最后返回，确保本模块自身不崩溃。这里笔者给出一段程序示例，各位读者可以好好体会一下。

例 3.7.3.1

```
//以下是一个申请获得服务器 IP 地址包的逻辑，此处仅为示例，请读者观察异常处理
bool CRelayAppClientPoolToken::GetAppServerIPGroup(void)
{
    bool bRet=false;           //所有返回码的初值首先设为异常，方便中途退出
    Linux_Win_SOCKET s=Linux_Win_InvalidSocket;
    s=m_pRelayProtocol->CreateUpTunnel( //此处动作不重要，关键是异常处理
        m_RelayAddr.m_szIP,m_RelayAddr.m_usPort);
```

```

        if(!SocketIsOK(s))
        {
            //异常常规处理手法：打印日志后，跳转到最后退出，注意，bRet 此时是 false
            XGSyslog("CRelayAppClientPoolToken::GetAppServerIPGroup():    relaye
connect fail! %s:%d\n", m_RelayAddr.m_szIP, m_RelayAddr.m_usPort);
            goto CRelayAppClientPoolToken_GetAppServerIPGroup_End_Process;
        }
        if(0>=m_pRelayProtocol->GetIPGroup(s, m_szDestID, &m_IPGroup))
        {
            //又一次异常处理的实例
            XGSyslog("CRelayAppClientPoolToken::GetAppServerIPGroup():    get    ip
group from relaye fail!\n");
            goto CRelayAppClientPoolToken_GetAppServerIPGroup_End_Process;
        }
        //核心代码逻辑...
        bRet=true;          //所有逻辑完成，最后设置成功返回标志
CRelayAppClientPoolToken_GetAppServerIPGroup_End_Process:
        //请注意，异常和非异常在此汇合，最终关闭 socket，防止资源泄漏。
        TONY_CLOSE_SOCKET(s);
        return bRet;        //返回处不关心 bRet 的值，值由前面的程序决定
    }
}

```

3.7.4 程序的易读性要求 goto

前文我们强调了要书写简单易读的代码，是无错化程序设计的基本要求。在笔者收到的关于 goto 的反对意见中，很多意见提出 goto 不是必须的，可以简单使用 if...else...来替换。因此，这里有必要讨论一下。

我们还是以上的例 3.7.3.1 来看，不过，我们换一种写法，完全以 if...else...实现

例 3.7.4.1

//以下是一个申请获得服务器 IP 地址包的逻辑，此处仅为示例，请读者观察异常处理

```

bool CRelayAppClientPoolToken::GetAppServerIPGroup(void)
{
    bool bRet=false;          //所有返回码的初值首先设为异常，方便中途退出
    Linux_Win_SOCKET s=Linux_Win_InvalidSocket;
    s=m_pRelayProtocol->CreateUpTunnel( //此处动作不重要，关键是异常处理
        m_RelayAddr.m_szIP, m_RelayAddr.m_usPort);
    if(SocketIsOK(s))
    {
        if(0>=m_pRelayProtocol->GetIPGroup(s, m_szDestID, &m_IPGroup))
        {
            //核心代码逻辑...
            bRet=true;          //所有逻辑完成，最后设置成功返回标志
        }
        else
        {
            XGSyslog("CRelayAppClientPoolToken::GetAppServerIPGroup(): get ip
group from relaye fail!\n");
        }
    }
    else
    {
        XGSyslog("CRelayAppClientPoolToken::GetAppServerIPGroup():    relaye

```

```
connect fail! %s:%d\n", m_RelayAddr.m_szIP, m_RelayAddr.m_usPort);
    }
    TONY_CLOSE_SOCKET(s);
    return bRet;
}
```

大家可以看到，随着 `goto` 的取消，程序代码段的书写开始因为 `if` 语句缩进，最中心的核心代码越来越靠右书写，在本实例中可能不明显，但一个商用的传输模块，很可能需要十几个条件都满足的情况下，才能真正实施服务，这样的话，程序的缩进已经很靠右了，往往超出屏幕的边界，最麻烦的是，最靠右，观察最不方便的，反而是最核心的处理逻辑，这给程序的易读性带来极大影响。

上述写法，会导致程序员很难通过简单的上下翻屏来浏览代码，这在小工程中，可能问题不大，但在几万行，十几万行的大型工程中，程序员想要靠肉眼的敏感度来分析代码，已经不太可能了。

还有一个问题也说明这么书写的危害，前面提过，很多编译器，优化时对大括号的层数有限制，`gcc` 一般是大括号超过 3 层，就拒绝优化，这可能会导致程序性能低下，应该尽量避免，因此，上述不断向右缩进的格式，一般建议不要采用。

这说明了 `goto` 的一个优点，减少大括号的层数，程序易读，且便于编译器自动优化。

3.7.5 break 为什么不能乱用

还有一种明确反对使用 `goto` 的意见，在循环跳出时，不需要使用 `goto`，可以简单实用 `break` 跳出，或者使用 `continue` 提前结束本轮循环。这在 5.3.3 节我们已经举例说明了，一般不建议这么做，这里笔者再补充一点理由，请大家看下面这个例子：

//例子一，希望连续跳出多重循环，几乎无法书写的程序

```
void Func(void)
{
    int i=0;
    bool bAllBreak=false;
    for(i=0;i<100;i++)
    {
        while(1)
        {
            //...
            if(bAllBreak) break;    //???希望一次跳出两重循环，这么写对吗?
        }
    }
}
```

//例子二，用 `goto` 轻松解决

```
void Func(void)
{
    int i=0;
    bool bAllBreak=false;
    for(i=0;i<100;i++)
    {
        while(1)
        {
            //...
            if(bAllBreak) goto Func_All_Loop_Break;
        }
    }
}
```

```

    }
}
Func_All_Loop_Break:
    return;
}

```

这个例子展示了，当多重嵌套循环内部，因为某种异常，需要一次跳出所有循环时，`goto` 的用法。这体现了 `goto` 的另外一个优点：**精确标定落点，使程序行为精准可控**。

我们前文已经说明，无错化程序设计的核心思想，并不是绝对没有 `bug`，而是使 `bug` 更加易于暴露，尽早暴露，降低 `bug` 解决的成本。因此，笔者在工程开发中，强调程序员不要怕打字，即使有隐式的系统约定使程序能够正常运行，也应该尽量使用显式代码，使程序的逻辑走向清晰可见。以达到尽早暴露 `bug` 的目的。

本小节的例子就说明了这一原理，当一个程序员习惯使用 `break` 的时候，默认的，`break` 仅仅跳出本层循环，这是语言隐含的约定，大家也会形成习惯，一旦遇到这种多重循环跳出问题，很可能程序员一个随手 `break`，就此留下 `bug`，并且，这个 `bug` 在编译器看来，是完全正确的程序逻辑，因此不可能报警，这个 `bug` 的查找代价将会非常高。

但很不幸，大型商用工程，多重循环比比皆是，同时，随时可能有异常处理，这种多重循环的跳出动作非常多，如果不使用 `goto` 精确标定落点，大家可以想象一下，`bug` 的数量会怎么样。

请记住，在商用工程设计中，尽量避免默认，程序员最好把你的意思明确表达出来。因为这段代码，明天肯能就不是你在维护了，接收的人，水平未必有你高，不一定看得懂的。就算看得懂，也不一定看得到。

3.7.6 goto 的常规使用手法

综上所述，我们可以达成一个共识，其实在 C 和 C++ 语言中，`goto` 语句之所有没有被取消，确实是因为它有不可替代的价值，只要谨慎地使用 `goto` 语句，可以解决程序实现中很多具体的困难，达到事半功倍的效果。

那么，如何谨慎使用 `goto` 呢？根据笔者经验，这里给出一点建议：

1、只能从大括号内部向外跳，永远不准跳进大括号，这是常识，`goto` 语句自己也是这么要求的，如循环体只能跳出，不准跳入，这里笔者就不多说了。

```

void Func(void)
{
    int i=0;
    if(...)
        goto Func_In_Loop;          //这种跳转不允许，不能直接跳进大括号内部
    for(i=0;i<100;i++)
    {
Func_In_Loop:
        //...
        if(...) goto Func_Loop_End; //这种跳转允许
    }
Func_Loop_End:
    //...
}

```

2、只能向后跳，严禁向前跳。这是笔者的经验，很多 `goto` 的滥用，都是从程序段落的末尾向开始跳，由于程序逻辑可能导致某个中间变量值的改变，这种跳转，通常都是灾难，应该严厉禁止。

```

void Func(void)
{
    int i=0;
Func_First_Action:
    if (0==i)    //在以后的跳转中，
        //...    //由于 i 值已经改变，因此，这个分支永远不会执行
        //...
    i++;
    goto Func_First_Action; //根据经验，这种向前跳转，应该严厉禁止
}

```

3、严禁使用 goto 实现死循环，死循环我们已经说过，用 while(1)来书写。例子同上。

4、根据 C++编译器的约定，goto 的标签后面，不能直接出现大括号，这时候需要在后面添加一个无意义的语句来使编译器不报错。通常是 return 或 continue 语句。

```

void Func(void)
{
    int i=0;
    for(i=0;i<100;i++)
    {
        if(...) goto Func_Loop_Continue;
        if(...) goto Func_End_Process;
        //...
Func_Loop_Continue: //由于编译器不允许标签后面直接有大括号
    continue;    //此处添加 continue 做占位。这也是 continue 唯一被使用之处
    }
Func_End_Process: //由于编译器不允许标签后面直接有大括号
    return;    //此处添加 return 做占位。
}

```

3.8 指针的使用原则

C 和 C++语言，最让人又爱又恨的，大概就是指针了。支持者说指针使 C 和 C++强大，灵活，可以使其做到很多汇编语言才能做的事情。反对者说，指针的滥用和费解，是导致这两门语言开发困难，程序员学习难度大，bug 率居高不下的关键因素。这么说都有道理。

笔者作为 C 和 C++程序员，不可避免地在工程中大量使用指针，根据笔者的经验，**指针可以用，但一定要谨慎地使用！**

3.8.1 商用数据传输常见的指针类型

在笔者的商用数据传输工程实践中，很多时候，需要定义 api，因为笔者的模块，最终的调用者很可能是别的程序员，必须先约定一套接口标准，然后大家分别开发，不管哪部分做了修改，只要接口标准不变，别人就不需要跟着变，这是合作式异步开发的关键。

在定义 api 的时候，笔者最为头疼的就是如何界定数据格式。商用工程的业务数据多种多样，各种类型都有，很难一个接口适应所有数据，最要命的是，如果这个 api 是提供给其他语言模块使用，各个语言对不同变量的表示又不一样，更加难以确定 api 的接口变量类型。

npi 的问题也差不多，跨操作系统通信开发时，各个平台还有个内存数字问题，比如一个 16 位的数字 0x00FF。Windows 执行的是 Intel 标准，低位在前，高位在后，内部表示为 0xFF00，而 Unix 和 Linux 则执行 DEC 的标准，高位在前，低位在后，仍然是 0x00FF。

学过 COM 的读者可能会有这个感觉，COM 接口其实可以视为 api 的一种，而 DCOM 接口，其实很类似笔者前文所述的 npi，大家可以发现，为了描述多种多样的数据类型，COM 接口提出了智能指针的概念，以一个庞大的变量包容类指针，再加上内部的数据类型描述，来描述数据。这造成了庞大的运行开销，不管是内存还是时间片，消耗都非常大。但没办法，为了用一套通用的数据类型描述所有数据，就这么麻烦。

大家可以想象，如果我们希望做一个通用的数据传输模块，首先这个数据类型的描述，就是个极其庞大的工程。因此笔者经过了长期的实践，制定了下面这个原则：

商用工程，特别是以数据传输为主的商用工程，所有的业务数据在交由传输层进行传输时，一律被视为二进制 char 型数据，调用者均有责任给出数据指针（char*表示）和数据长度（int 表示），传输层负责传输，接收方收到后，根据业务自行理解和转换。因此，商用数据传输工程所有的数据，均以一个 char 型指针和一个 int 长度表示。

通过这样的方法，笔者简化了商用工程所需要处理的数据类型，并且，每个数据均视为单字节的 char 数据，这在各个语言和各个平台上均不会出现歧义，这样做出来的 api 和 npi 接口，简单实用，可以适应任何应用。

因此，笔者的工程中，使用得最多的指针类型，是 char*。同时，这也隐含约定，商用数据传输工程，在数据传输时默认的数据位宽，都是一个 Byte(8bits)。

这样做有个潜在的好处，相当于我们数据传输模块的数据位宽被默认限制为 1Bytes，这对于很多根据位宽进行的计算，如 strlen，sizeof 等，能求出比较准确的数值，不会因为数据类型的位宽差，导致计算出错，引入潜在的 bug。

还有一种指针类型也比较常见，void*，这主要是用在函数构型的参数设计中，根据 C 语言的规定，当一个函数接收 void*时，实际上可以把任何类型的指针直接作为参数调用，因此比较常用。

至于其他的指针类型，一般是各种类指针，结构体指针，基本上根据需要定义使用，一般不具备普适性，就不再赘述。

3.8.2 不要使用两个以上的*号

其实笔者个人认为，很多时候，C 和 C++语言的指针设计，遭人诟病之处，主要就是指针的滥用，最突出的就是多个*号的存在，即**指向指针的指针**。

这确实是一个很费解的设计，笔者认为，程序员毕竟是人，人的脑子是有限的，当一个逻辑关系太过于复杂的时候，人往往会记不住，脑子会很混乱，这样写出来的代码，很容易产生 bug。

而这类指向指针的指针，就是典型的复杂逻辑。就笔者自己的情况，不用多了，一般有三个到四个*号，笔者自认为就记不住了，也没有办法掌控程序的质量。因此，笔者在商用工程实践中，一般在团队内部规定，严禁使用两个以上的*号。

有趣的是，往往项目启动会议上，笔者的这个规定一出，程序员立即哀鸿遍野，然后就是气势汹汹地反弹，程序员举出各种样例来说明多重*号的必要性，最多的例子就是如果修改传入一个变量的值，以及二叉树等复杂数据结构的实现。

对此，笔者一概不予理睬。原因很简单，这些地方都有替代的方法，多重*号不是必须的和唯一的解决方案。

一般说来，一个函数作为参数的使用者，应该避免修订传入的参数的值，这类情况在系统设计时就应该予以避免。参数作为计算的参与者，如果再负担数据传递的任务，其实就是前文所说的变量转义，很容易误导程序员，造成 bug。

不过，笔者也同意，很多时候，一个函数需要返回多个参数时，将被迫使用参数修改这一技巧，这个时候，完全可以使用&传址调用来实现，没有必要滥用指针。

当然还有一种情况，，有时候我们需要函数修改的参数值，本来就是指针类型，这其实已经涉及到多重*号了，对此，笔者统一的建议是，请使用一个结构体来封装需要修改的数据，函数内部修改**结构体指针指向的结构体数据**的值，也不要多重*号。这在后文的结构体传参设计中，有更详细的介绍。

请大家相信，笔者做了这么久的商用工程，从来没有使用过两个以上的*号，也成功完成了所有的工程项目，C 语言的这个设计，不是必须的，请尽量回避使用，避免 bug。

3.8.3 指针不能参与四则运算

在 C 语言的世界里，其实有一个数据类型无处不在，就是 int 整型。我们仔细分析一下可以发现，int 是整型，bool 也是整型，所有的指针，还是整型数据。因此，C 语言有些奇怪的写法，不准确，但大家又都在用，比如下面的例子：

```
while(1) //这是把整数当 bool 用
if(!nCount) //这也是把整数当 bool 用
if(!pBuffer) //这是把指针当 bool 用
```

作为一个严谨的语言，上述的写法其实并不是很准确，严格说应该避免，但是，C 语言毕竟发展这么多年了，很多写法都约定俗成了，不这么写，读的人反而费劲，因此就保留下来了。甚至到了 C++，也尊重了这一习惯。

但这类写法，故意混淆了各个变量类型的差别，往往给程序员一个误导，数据都是一样的，可以随意进行计算，这就是很坏的习惯了，严格说应该禁止。

其中最突出的例子，就是很多程序员喜欢把指针作为整数参与计算，笔者自己就干过利用两个指针的差值来求数据段的长度的事情。

```
int GetDataLength(char* pStart, char* pEnd)
{
    return ((int)pEnd) - ((int)pStart);
}
```

这段代码，任何 C 或 C++ 编译器都会认为是正确的，编译无误，运行也正确。但是，大家想过没有，指针的真实含义，一个指针，是指向某个数据区的值，两个不同的指针，指向两个不同的数据区，各自代表不同含义，二者没有任何关系，怎么能参与计算呢？

指针就是指针，是用来指示数据的工具，它不是数据，用它进行四则运算，是没有任何意义的。

这好让大家比较，时速 200 公里/小时和质量 130 吨相加会有什么结果一样，二者没有任何关系，单位都不一样，无法计算。

在实际的工程开发中，我们也有可能遇到，两个指针，可能是两个进程空间的指针，二者本来就**不是连续编址**，没有线性可比性，上述的计算直接就是错误。

所以，上述函数的设计，看似一个通用性很高的功能调用，非常有用，但笔者通常把它视为 bug 之源，只要它存在一天，程序员就可能滥用，这种函数应该第一时间删除，逼着程序员改用其他安全的算法，以避免潜在的 bug 隐患。

笔者在所有的工程项目中，一般强调：“**严禁变量转义，严禁使用指针参与四则运算！**”。

当然，有一种特例是允许的，C 程序员通常把缓冲区和数组故意混淆，因为这样可以带来程序最大的灵活性。

比如，一个数组 a[10]，我们可以用 a[0] 访问第一个单元，也可以使用 *(a+0) 来访问，这在很多关于字符串的词法分析中很有用。

笔者一般把这种写法视为一种特例予以允许，但仅此一种，即**一个指针，允许和一个整数做一次加法计算**，以便求的缓冲区和数组的偏移量。

指针不准参与四则运算这一原则，还在后文所述的 32 位改写成 64 位工程的例子中，起到了意想不到的作用，这在后文有详述。

3.9 使用结构体的技巧

本小节应该算是指针使用原则的补充，详细解说如何使用结构体来传参。不过，在笔者后续的使用中，发现结构体传参不仅可以避免多重指针的滥用，还有其他的很多优势。

3.9.1 结构体传参的必要性

在我们开发的过程中，函数构型的设计无处不在，在设计函数时，参数的设计往往是一个大问题，尤其是针对某些功能较为复杂模块的主要入口函数，其参数非常多，程序员操作起来非常复杂。并且，书写方法繁琐，很容易出错。

举个例子，比如 Win32API 中窗口的建立函数 `CreateWindowEx`，其函数构型就非常复杂：

```
HWND CreateWindowEx(DWORD dwExStyle,
                    LPCTSTR lpClassName,
                    LPCTSTR lpWindowName,
                    DWORD dwStyle,
                    int x,
                    int y,
                    int nWidth,
                    int nHeight,
                    HWND hWndParent,
                    HMENU hMenu,
                    HINSTANCE hInstance,
                    LPVOID lpParam);
```

大家可以想象一下，这么多的参数，如果再不使用一些有意义的变量名，很可能函数调用时的可能是这样的：

```
HWND h1=CreateWindowEx(de,p1,p2,ds,x,y,w,h,hf,hm,hi,pp);
```

大家觉得这样写好不好？容不容易出错？因此，一般的工程化开发建议，如果一个函数由于业务需要，有很多参数，建议把参数总结到一个结构体中，用一个结构体指针来传参，可以避免很多潜在的 bug 隐患。

同样上面的例子 `CreateWindowEx`，如果笔者来设计这个函数，大约会写成如下形式：

```
typedef struct _CreateWindowEx_Param_
{
    DWORD        m_dwExStyle    ;           //请注意，参数与原文一一对应
    LPCTSTR      m_lpClassName  ;
    LPCTSTR      m_lpWindowName ;
    DWORD        m_dwStyle      ;
    int          m_x             ;
    int          m_y             ;
    int          m_nWidth        ;
    int          m_nHeight       ;
    HWND         m_hWndParent    ;
    HMENU        m_hMenu         ;
    HINSTANCE    m_hInstance     ;
}
```

```

        LPVOID        m_lpParam        ;
    }SCreateWindowExParam;
HWND CreateWindowEx(SCreateWindowExParam* pParam);

```

这样，虽然设计这个接口的时候麻烦点，但是，别的程序员在调用我的接口时，会使用如下格式调用，几乎不可能错误：

```

void Func(void)
{
    HWND hMyWindows=null;
    SCreateWindowExParam CreateWindowExParam;
    CreateWindowExParam.m_dwExStyle    =...
    CreateWindowExParam.m_lpClassName  =...
    CreateWindowExParam.m_lpWindowName =...
    CreateWindowExParam.m_dwStyle     =...
    CreateWindowExParam.m_x           =...
    CreateWindowExParam.m_y           =...
    CreateWindowExParam.m_nWidth      =...
    CreateWindowExParam.m_nHeight     =...
    CreateWindowExParam.m_hWndParent  =...
    CreateWindowExParam.m_hMenu       =...
    CreateWindowExParam.m_hInstance   =...
    CreateWindowExParam.m_lpParam     =...
    hMyWindows=CreateWindowEx(&CreateWindowExParam);
}

```

这是结构体传参的最大好处，把所有的变量在调用前，以清晰的格式列举出来，利用匈牙利命名法，严谨地提醒程序员，每个变量的类型，名称含义，分别予以赋值，这样做虽然有点麻烦，但程序员出现人为错误的可能性大为减少。

这里可以看出无错化程序设计的另外一个思路，通过严谨的设计，不仅仅保证自己的程序尽量减少 bug，也强迫自己的调用者，不再出现 bug。

另外，包括 VC 的 IDE 在内的很多编辑器都有代码自动补全的功能，上述书写虽然看似麻烦，但其实输入时候并不困难，基本上都是编辑器的工作。

3.9.2 预防多重指针的隐患

这在前一小节已经说明，结构体传参，可以避免多重指针的使用，这里我们来看一个实例：

```

//这是申请一个缓冲区的例子，里面用到了两重*号
bool MallocABuffer(char** ppBuffer,int nBufferSize)
{
    *ppBuffer=(char*)malloc(nBufferSize); //此处修改传输的指针的值
    //...
}
//这是利用结构体传参回避了多重*号的写法
typedef struct _MallocABuffer_Param_
{
    char* m_pBuffer;
    int m_nBufferSize;
}SMallocABufferParam;
bool MallocABuffer(SMallocABufferParam* pParam)
{

```

```
//原则：修改指针指向数据的值，但不要修改指针本身
pParam->m_pBuffer=(char*)malloc(
    pParam->m_nBufferSize);
//...
}
```

大家看到，通过这个简单的变换，我们已经成功规避了多重*号的滥用。因此，笔者可以论证，所有的 C 或 C++ 工程，其实可以不必使用多重*号的，这样可以避免很多潜在的 bug 隐患。

3.9.3 32 位到 64 位移植

结构体传参的另外一个好处，确实是笔者当初没有设想到的，但近期的一个工程项目，却印证了这种方法的优越性。

笔者在前不久接到一个任务，由于目前 64 位 CPU 已经上市大量销售，很多客户购买的新 PC 服务器平台，都是 64 位的，并且 Linux 也已经推出 64 位版本，因此，需要将以前做过的一个服务器集群项目，由 32 位改写成 64 位，以适应新的系统。

这个任务难度倒是不大，但关键非常繁琐，由于在 64 位 Linux 下，gcc 的 int 还是 32 位，但是 long 和所有的指针，都改变成 64 位。这要几万行代码逐一检查修改下去，工作量不可想象。并且，还可能有很多 bug 需要修改。笔者不禁冒出一身冷汗。

但仔细分析下来，笔者却发现情况没有那么悲观，首先，由于笔者前面坚持的指针不准参与四则运算的原则，笔者的工程中，几乎不存在指针运算，即便是有，也是前文提到的特例，即字符串缓冲区的加法计算*(p+n)，由于此时的指针已经是 64 位，根本没有溢出危险，因此，无需考虑。

还有就是笔者习惯使用结构体传参，由于严格禁止了指针的任意修改，换句话说，所有的函数内部都不会修改指针参数的值，所有的传输参数几乎都是只读的，因此，指针的位数变化，根本不会带来任何影响，程序天生就是兼容 64 位的。仅需处理几个 long 型变量即可。

笔者带着试试看的心情，Review 了一遍代码，果然不出所料，仅仅是修改了大约 5、6 处地方，将原有的 int 改为 long，整个修改工程即宣告完工。测试没有 bug。

这件事笔者得出一个结论，C 和 C++ 无错化程序设计，由于其严谨的风格和严厉的规则限定，确实可以避免很多潜在的 bug，并在未来需求改变时，能做到自动适应，将 bug 防患于未然。

3.9.4 弹性内存使用需要结构体传参

在前文“粘合类”的小节中，这个话题已经讲过一次，不过，这个关键点由于非常重要，我们这里结合结构体，再讲解一遍，希望加深大家的理解。

在我们用 C++ 开发的过程中，类、对象是很方便的一个概念，也是面向对象程序 (OO) 设计的基础。

我们知道，正常情况下，每一段程序都是有初始化代码、业务执行体、结束代码构成的，为了准备一个计算，我们需要申请一些资源（如内存），准备一些变量，然后开始计算，计算结束，我们需要释放资源和变量，这构成了面向过程 (OP) 的程序设计基础。

而 C++ 的对象设计，使用构造函数和析构函数来完成初始化代码和结束代码，很多时候，能把程序员的注意力集中到要进行的计算本身，同时，在复杂的调用模型中，也不必机械地书写大量的初始化模块和结束模块的调用代码，这使程序显得很简洁，并且开发效率很高。

但这样一来，有一个小小的缺陷，就是 C++ 不能再使用 C 的动态内存申请机制来处理对象，因为 C 的 malloc 和 free 函数，没有自动调用构造函数和析构函数的功能，必须代之以 new 和 delete。

这在常规的程序设计中，没有任何问题，但在服务器和嵌入式设备开发中，由于有 7*24 小时的运行需求，对内存碎片很敏感，当我们需要管控所有的内存操作时，会出现问题。

通常情况下，服务器或者嵌入式设备，都有内存池的概念，即应用层在操作系统之上，再建立一套内存管理机制，通过对内存块尺寸的取模，使内存块达到高度可重用，进而减少对内存的申请和释放动作，实现运营安全。在后文介绍的库中，有专门的内存池模型供大家参考。

但在 C++ 中，由于对象的申请和释放机制需要 new 和 delete 来支持，不是简单的内存块，因此，不能用上述方式来处理对象，那么，如果一个应用程序，使用中申请和释放了大量的对象，则内存碎片问题仍然存在，给程序带来安全隐患。

为了解决这个矛盾，笔者在无错化程序设计方法中，总结出一个 C++ 类的使用原则，这里供大家参考：

所有的类成员变量，笔者分为业务数据和内部服务数据两类，分别处理。

业务数据主要是为了完成类实现功能的相关数据，如一个传输模块的传输数据缓冲区，这些数据不是它的，是用户委托他传输的业务数据。

内部服务数据，则是为了实现这个业务功能，需要准备的一些临时变量，比如上述传输模块传输中申请的 socket 接口，传输字节数计数器等，这些是它本身内部的数据，外界不知道，也不关心。

那么，对于业务数据，由于长度变化很多，经常需要动态内存申请，建议采用结构体管理，并使用上述内存池原则进行管理，而内部服务数据，则尽量采用静态成员变量，不要使用动态内存申请，在程序运行浮动栈上由系统管理。

并且，在系统设计时，尽量采用静态对象，减少动态对象的使用，尤其是注意保证对象的抽象性，提升对象重用度，规避运行期对象的频繁申请和释放，减少内存碎片。

举个例子，这里是一个传输模块，大家可以看到对数据的界定和结构体的使用：

```
#define MAX_PACKAGE_LENGTH 256 //最大报文长度
typedef struct _CRW_DATA_
{
    char m_szData[MAX_PACKAGE_LENGTH]; //这里定义静态缓冲区
    int m_nDataLength; //内置数据的真实长度
} SCRWData;
const ULONG SCRWDataSize=sizeof(SCRWData);
class CRW
{
public:
    CRW()
    {
        m_pData=(SCRWData*) malloc(SCRWDataSize); //这可以理解为内存池申请
        if(m_pData)
        {
            m_pData->m_nDataLength=0; //初始化变量
        }
    }
    ~CRW()
    {
        if(m_pData)
```

```

        {
            free(m_pData);                //这利益理解为内存池释放
            m_pData=null;
        }
    }
    int Send(char* szData,int nDataLength) //业务函数
    {
        if(!m_pData) return -1;          //防御性设计
        //...
    }
private:
    SCRWDData* m_pData;                  //以结构体管理业务数据
    Linux_Win_SOCKET m_nSocket;          //内部服务数据使用直接变量
};

```

大家可以看到，结构体中的数据是业务相关，那么，这个结构体可以通用，另外的一些类或模块，也可以使用这个结构体，这样，不同的模块，在同一笔业务中，就具有相同的表述，使用起来非常方便。

另外一方面，由于关键的业务模块使用结构体封装，在内存池支持下，可以在很多场合任意使用动态内存，而不是遇到一笔数据，必须 new 一个 CRW 对象来存储数据，这在减少内存碎片的需求上很有用。在后续的应用代码中，我们还能看到类似的实例。

3.9.5 网络传输协议，需要结构体传参

本小节算上一节的补充。我们知道，网络传输协议中，主要分为两大类，一类是文本信令，另外一类是二进制信令。文本的姑且不论，我们可以考虑 Json、xml 等描述协议来解决，但二进制信令这块，通常没有一定之规，这就造成了很多实践上的混乱，很多 bug 也由此产生。

而笔者认为，二进制信令，最好的方式莫过于使用结构体来完成，很多时候，甚至需要结构体和联合体共同完成。如下面的信令实例：

```

#define MAX_ID_LENGTH 256                //最大字符串长度，用户名和密码都用这个
typedef struct _SIGNAL_LOGIN_            //用户登陆包
{
    char m_szUserName[MAX_ID_LENGTH];    //用户名
    char m_szPassword[MAX_ID_LENGTH];    //密码
}SSignalLogin;
const ULONG SSignalLoginSize=sizeof(SSignalLogin); //结构体尺寸
typedef struct _SIGNAL_MSG_              //消息报文
{
    char m_szTimeStamp[MAX_ID_LENGTH];   //时间戳
    char m_szFromUser[MAX_ID_LENGTH];    //发送方用户名
    char m_szToUser[MAX_ID_LENGTH];      //接收方用户名
    char m_szMsg[MAX_ID_LENGTH];         //消息本体
}SSignalMsg;
const ULONG SSignalMsgSize=sizeof(SSignalMsg);    //结构体尺寸
#define RW_SIGNAL_LENGTH sizeof(char) //以一个字节作为收发信令
#define RW_SIGNAL_DATA_BEGIN(p) (p+ RW_SIGNAL_LENGTH) //数据区起始点计算
#define RW_SIGNAL_LOGIN 0x01          //Login 的信令
#define RW_SIGNAL_MSG 0x02            //Msg 信令
typedef struct _RW_SIGNAL_PACKAGE_

```

```

{
    char m_cRWSignal;           //这是供接收方检索报文类型使用的收发信令
    union                       //联合体，内存使用紧凑
    {
        SSignalLogin    m_SSignalLogin; //Login 的数据解释
        SSignalMsg      m_SSignalMsg;   //Msg 的数据解释
    };
}SRWSignalPackage;

```

这是笔者常用的二进制信令构造手法，大家可以看到，这个协议包含两条信令，Login 和 Msg，这是笔者假设的一个在线聊天系统的信令。这里面有几个关键点，请大家关注：

- 1、每个信令独立使用结构体封装。
- 2、在主报文 SRWSignalPackage 中，利用一个字节的报文类别，来区别是什么信令
- 3、后续数据段采用联合体节约内存空间。

这种设计，使信令的设计简单，精准，并且各个数据边界均可控，为后续的收发活动打下基础。并且，这种结构体信令本身，就可以作为参数传递，使不同的模块针对同一笔业务，仅有一种表现形式，简化程序的数据访问复杂度，也降低 bug 隐患。

综上所述，结构体即使到了 C++ 阶段，仍然是一种很有用的数据组织方式，各位读者在以后的设计中，利用本书所示的技巧，使用结构体进行开发，会起到事半功倍的作用。

3.10 使用宏的建议

简单的说来，宏是 C 语言下的一个工具，宏的含义是在编译器预处理时，将程序中的宏定义，采用文本替换的手段，直接替换为指定的程序段落，在编译器看到的是替换后的结果，因此，这个工具，更像一种程序编辑的简化手段而已。

而在 C++ 语言中，由于面相对象的强类型要求，已经不再建议程序员使用宏这个工具。但由于笔者有较深的 C 开发背景，因此开发中还是比较习惯使用宏的，这里也向大家展示一些使用宏来降低 bug 的技巧。

但这里笔者强调一句，本小节的技术，仅为建议使用，读者如果要遵循严格的面面向对象程序设计规范，可以不采用。

3.10.1 宏的几大作用

在 C 语言中，宏主要有两个作用，第一是定义常量，第二是固化计算。而在笔者看来，还有第三个作用，即简易的函数式编程。

1、定义常量，主要目的是为了简化将来的修改操作，如我们将 PI 定义为圆周率 3.1415926，在程序中需要用到圆周率的地方，全部使用 PI 代替，假如以后圆周率有了新的改变，我们无需在程序中每个调用点做修改，仅需简单修改这个宏定义就够了。

```
#define PI 3.1415926
```

2、固化计算，这主要是一些计算性工作中，对于某个小的计算式，可能需要在运行期根据测试情况做出调整，又或者有些计算关系，大家规定这么用，通常就使用宏来固化，每个调用点使用宏表示，这也是为了将来调整方便。

例如前文中协议信令例子，我们定义一个传输缓冲区，第一个字节是信令，第二个字节开始是数据，则我们可以用这个宏来固化计算。

```

//数据区起始点计算
#define RW_SIGNAL_DATA_BEGIN(p) (p+ RW_SIGNAL_LENGTH)

```

这样，当我们以后修改缓冲区偏移定义时，仅仅简单修改这个宏即可。

3、目前函数式开发很热门，主要是函数式开发中，很少用到变量，所有的计算中间值基本上都是固化的常量，因此，后续开发基本上采用只读方式来访问中间值。我们知道，只读是线程安全的，因此，函数式编程天生就具有跨线程的安全优势，很适合开发并行计算应用。

很遗憾的是 C 和 C++ 作为传统语言，并没有对函数式编程有更好的支持，因此，某些时候，当笔者需要用到函数式编程时，宏几乎是唯一的选择。而宏不检查变量类型的特性，此时也正好变成了优势。

比如我们知道一个公式，利用半径求得圆的面积，这在函数式编程中，就是定义一个计算关系， $P \cdot r \cdot r$ ，这如果利用宏来实现，将非常方便。

```
#define PI 3.1415926
#define Get_Circular_Area(r) (PI*((double)r)*((double)r))
```

请大家注意，此时传入的 r 参数，既可以是整型，也可以是浮点型，都不会出错。这类利用宏的固化计算能力，实现函数式编程，是 C 语言的一个较为深入的用法。

3.10.2 C++的建议

不过，在 C++ 内，由于强类型检测需要，已经不再建议使用宏这个设计，这个见仁见智，笔者这里仅提供建议而已。

在 C++ 中，常量的表示一般建议使用 `const` 完成，而函数型宏，建议使用 `inline` 函数。不过，这样做可能需要用到函数重载，以适应多种变量类型。如上例变为以下例子，就显得很不方便。

因此，笔者既不说宏很好，也不说不好，各位读者可以根据自己的喜好来选择。

```
const ULONG SSignalMsgSize=sizeof(SSignalMsg); //C++常量定义例子
inline double Get_Circular_Area(int r)          //针对整型数的计算
{
    return (PI*((double)r)*((double)r));
}
inline double Get_Circular_Area(double r)      //针对 double 类型数的计算
{
    return (PI*((double)r)*((double)r));
}
inline double Get_Circular_Area(float r)       //针对 float 类型数的计算
{
    return (PI*((double)r)*((double)r));
}
```

3.10.3 编译宏—跨平台开发

在很多时候，我们需要用到编译宏。在跨平台开发中，各个平台，总有一些细节的差异不太一样，为了维护程序的一致性，需要针对某些特定平台做一些特定的定义，而这些定义，最终被表现为编译宏。

不同的 Makefile，不同的 IDE，都有一些编译宏，以界定自己的平台属性，常见的有 WIN32 这个编译宏，一般笔者常用来界定 Windows 相关代码和 Linux 相关代码。在代码中，笔者经常书写如下的段落：

```
#ifdef WIN32
//Windows Code...
```

```
#else // not WIN32
//Linux Code...
#endif //WIN32
```

编译宏各个平台有很多定义,但作为跨 Windows 和 Linux 平台的程序开发,这个 WIN32 定义已经能满足大多数时候的需求,各位读者有兴趣的话,可以自行搜集类似的定义。Mac 的操作系统,基本上也可以视为一种 Unix Like 的操作系统,因此,上述代码也基本适用。

3.11 回调函数设计方法

在工程设计中,尤其在底层库设计的时候,很多时候,库的开发者并不能预测今后使用这段代码的程序员需要这个函数做具体什么工作,这时候,就需要使用回调设计。

举个例子,当我们一个类,内部掌握一个私有数组,对外提供回调遍历方法,外部调用者无需关心内部数组的具体组织形式,以及各种复杂的遍历手段,仅仅关心数据本身。这是一个很好的数据封装设计,在工程中常用。

比如 Win32api 就提供了大量的枚举遍历调用模型,如当前音频设备,当前摄像头等,都是系统提供枚举方法,应用程序使用枚举回调,查找自己需要的设备。

尤其是这个类有多线程访问需求的时候,由于锁操作的复杂性,因此笔者在前文建议大家参考资源锁的概念,在类内部封装和收拢所有复杂的锁操作,避免二元性动作带来的隐患。

这个时候,遍历回调的优势更加明显,数据的使用者无需关心加锁和解锁的细节,只关心数据是安全可用的。

C 和 C++ 都提供这类回调支持, C 的建议是使用函数指针,回调实现, C++ 则通过对基类的继承,对基类中虚函数的再设计来实现。不过,根据笔者经验,在这点上, C 的方式比 C++ 方式要轻灵,并且更加灵活。因此,在笔者的工程开发中,一般使用回调函数设计,不太使用虚函数机制。

回调函数其实就是函数指针的应用,在 C 中,一切数据均可以指针化表示,函数本身其实也可以,当我们以正确的构型调用一个函数指针时,其效果和直接调用函数本身,完全一样。

另外,由于现代操作系统的 C 亲密性,很多操作系统级的 api 设计都可以看到回调函数,比如我们常见的线程函数,甚至进程本身,其实都是操作系统的回调函数, `beginthread` 这类启动线程的调用,一般就是把指定的线程函数指针,在系统的线程表中,注册一个新的表项,系统下一轮时间循环,自动会根据这个表项,回调该指针,进而实现应用程序线程对时间片的获取。并且,这个过程,一般都是纯 C 的,和 C++ 无关。

从某种意义上说,现代并行计算,是建立在 C 的回调模型上的。作为商用工程的程序员,对于回调函数,应该有很深入的认识,并能熟练应用。在本书所展示的工程库中,更是大量使用回调模型,尤其是线程池等关键模块,几乎完全建立在回调模型之上,请各位读者关注。

回调函数的设计非常简单,不过,这里面首先要搞清楚两个身份,一个是回调函数的设计者,一个是使用者,但二者都是程序员。

大家别笑,笔者在项目讨论中,经常发现有程序员把这两个身份搞混,说出来的话鸡同鸭讲,谁都听不懂。在后文中,一律使用**回调模型设计者**和**使用者**来区分这两个身份。

3.11.1 回调模型设计者

作为回调模型的设计者，首先需要定义一个回调函数构型，因为 C 语言就算再灵活，也需要知道函数原型是怎样的，才能确保使用者是正确调用，避免崩溃。

这里我们举一个简单例子，即后文中的 **LowDebug** 模块，有时候需要把自己的输出信息提供给上层应用程序，供输出转向使用，这在后文有完整源代码，此处我们仅提供示例。我们先看看回调函数构型：

```
typedef void (*_APP_INFO_OUT_CALLBACK)(char* szInfo,void* pCallParam);
```

大家请注意，这里面体现出几点关键：

1、typedef，这是我们显式定义一种新的变量类型，这个变量类型，就是这一个回调函数指针的类型。以后使用这个指针的设计者和使用者，都可以使用

`_APP_INFO_OUT_CALLBACK` 这个变量类型来定义自己的指针变量。

2、本回调函数使用 `void` 作为返回值，是因为这个特殊应用。其实很多时候，有个约定，一般回调函数使用 `bool` 作为返回值，这在某些循环遍历的场合，当使用者感到自己的数据已经找到，循环无需继续，可以返回个 `false`，设计者就知道，可以不再循环了。这体现出使用者不是完全被动的接受回调，也可以通过返回值影响回调发起方的逻辑。

3、`char* szInfo` 这是业务数据，这里不再细说。

4、`void* pCallParam`，这个非常关键，所有回调函数的设计者，一定要帮助使用者传递一根 `void*` 的指针，并透传到每一次回调调用中。这个原因下一节会详细讲。

这里，我们来看 **CLowDebug** 的类关于设计回调模型的书写方法

```
class CTonyLowDebug
{
public:
    CTonyLowDebug(char* szPathName,
        char* szAppName,
        bool bPrint2TTYFlag=false,
        //构造函数传入回调函数和参数，可以是 null
        _APP_INFO_OUT_CALLBACK pInfoOutCallback=null,
        void* pInfoOutCallbackParam=null);
    //保存在对象内部，方便 Debug 等功能函数调用
    _APP_INFO_OUT_CALLBACK m_pInfoOutCallback;
    void* m_pInfoOutCallbackParam;
};
//我们来看看构造函数的回调指针赋值
CTonyLowDebug::CTonyLowDebug(char* szPathName,
    char* szAppName,
    bool bPrint2TTYFlag,
    _APP_INFO_OUT_CALLBACK pInfoOutCallback,
    void* pInfoOutCallbackParam)
{
    m_pInfoOutCallback=pInfoOutCallback;           //回调函数指针保存
    m_pInfoOutCallbackParam=pInfoOutCallbackParam; //参数指针保存
    //...
}
//然后，我们来看看设计者如何调用回调函数
int CTonyLowDebug::Debug2File(char *szFormat, ...)
{
    //...
```

```

    if (m_pInfoOutCallback)           //标准写法，先判断指针有效性
    {
        m_pInfoOutCallback(szInfoOut, //像函数一样调用
            m_pInfoOutCallbackParam); //这里在帮助透传指针
    }
    //...
}

```

大家可能关注到了回调函数设计的特点：

- 1、先定义回调函数原型，顺便定义一个新的指针变量类型。
- 2、设计者以该回调函数指针变量类型定义新的变量，实现参数传递和数据保存。
- 3、调用前先检查指针有效性，避免跳到空指针处，造成崩溃。

3.11.2 回调模型使用者

作为使用者来说，如果回调函数设计者均基于上述方法设计，其调用程序设计也可以形成简单规律和套路。

使用者首先必须以回调函数构型构建一个函数，这就是将来的回调函数实体，设计者的模块会跳至此处运行。使用者在这个函数内部，直接使用传来的变量 `szInfo` 即可，这就是每次 `Debug` 模块输出的字符串。

```
void ApplicationInfomationOutCallback(char* szInfo,void* pCallParam);
```

但有一点注意，如果是 C 里面，可以这样直接声明和实现函数即可。但在 C++ 的类中，不能这样直接写。这是由于 C++ 的编译器，为每一个类成员函数，提供了一个默认的隐含参数指针 `this`，指向本次实例化的对象，其类型就是这个类本身。因此，如下所述，这个函数就不对了。

```

class CTestTonyLowDebug
{
private:
    void ApplicationInfomationOutCallback(char* szInfo,void* pCallParam);
};

```

此时的回调函数原型，由于是类成员函数，有隐含指针，因此相当于如下原型

```

void ApplicationInfomationOutCallback(
    CtestTonyLowDebug* this,    //这是 C++编译器在编译时强行添加的
    char* szInfo,
    void* pCallParam);

```

这时，我们再和回调函数原型比较，发现多了一个 `this` 指针。

```
void ApplicationInfomationOutCallback(char* szInfo,void* pCallParam);
```

这说明，两个函数不是一个构型，函数指针类型不匹配，调用将会失败。

因此，所有的回调函数，一旦写在类里面，必须用 `static` 修饰为静态成员函数。

```

class CTestTonyLowDebug
{
private:
    //请注意这里的 static 修饰
    static void ApplicationInfomationOutCallback(
        char* szInfo,void* pCallParam);
};

```

C++ 规定，对于静态类成员函数，将不提供隐含的 `this` 指针，因此，函数的编译后本体和书写时的声明完全一样，这样就可以把这个函数作为回调函数。

但这随之带来另外一个问题，就是没有了 `this` 指针，使用起来很不方便。我们知道，C++的面向对象设计中，其对象的核心定义就是“一批数据和针对该数据的所有方法的集合。”，这是面向对象程序设计的精髓。

因此，一个类的成员函数方法，一般说来，都和这个类实例化的对象所包含的数据密切相关，程序中需要不断访问本对象的成员变量或其他成员函数，也就需要频繁访问本对象指针 `this`。C++编译器为类成员函数自动添加 `this` 指针，目的就在与支持此应用。如这个例子：

```
class CTestClass
{
    void Func(void)
    {
        m_nClassVar=0;
        //这句话编译后相当于
        this->m_nClassVar=0;    //this 指针，由编译器确定对象地址后决定
    }
    int m_nClassVar;
};
```

由于我们把回调函数写成静态的，就没有 `this` 指针了，这种对内部成员的访问自然也就失效，这时候的回调函数，可以说和本类对象没有一点关系，写在类里面和写在类外面没有差别。这当然这不是我们想要的。

我们希望，一个回调函数，能够紧密结合到其服务的类中，为其数据服务，从程序源代码上，以及业务逻辑上，都可以看做一个整体，另外，还必须满足回调函数的静态需求，不能有 `this` 指针。大家看复杂不复杂？

正是由于有这些限制和需求，最后，笔者给自己规定了一个原则：“**回调函数设计者有义务为使用者透传一根 `void*` 的参数指针**”。这就是上一小节，透传的参数指针的由来。

有了这根指针，我们当然可以做很多事情，最简单的就是传递本对象的指针给回调函数本身，以替代 `this` 指针。

因此，在本文的例子中，如果我们的 `Test` 类是如下构型，内部有一个 `LowDebug` 的对象指针变量，并需要利用其回调模型拦截输出数据。

```
class CTestTonyLowDebug
{
public:
    CTestTonyLowDebug();
private:
    static void ApplicationInfomationOutCallback(char* szInfo,
        void* pCallParam);
    CTestTonyLowDebug* m_pDebug;
};
```

则其构造函数和回调函数的实现中，应该以以下方式书写和调用：

```
CTestTonyLowDebug::CTestTonyLowDebug()
{
    m_pDebug=new CTestTonyLowDebug(...,          //前面的参数略
        ApplicationInfomationOutCallback, //这是把本类的回调函数指针传入
        this);                                //注意这里，this 指针被作为 void* 参数指针传入
}
void CTestTonyLowDebug::ApplicationInfomationOutCallback(
    char* szInfo,
    void* pCallParam)
```

```

{
    //由于本类的 this 指针已经被通过这个 void* 的参数指针透传进来，
    //此处我们可以做强制类型转换，获得 this 指针。
    //为了和 C++ 原有的 this 区别，提醒程序员这是传入的指针，笔者统一使用 pThis 命名
    CTestTonyLowDebug* pThis=(CTestTonyLowDebug*)pCallParam;
    pThis->...    //利用这个 pThis 指针，可以访问本类的成员变量和成员函数
}

```

大家可以看到，由于设计者设计了透传参数指针的机制，调用者可以很方便地将 this 指针传给回调函数，回调函数再通过简单的指针类型转换，即可获得和普通成员函数相同的权利和功能，实现本对象内部成员的访问。

3.11.3 参数传递的常规手法

上文中，设计者实现 void* 的参数指针透传机制，使用者传递本对象指针，仅仅是回调函数众多参数传递需求中的一种。实际使用中，当然远远不够。

很多时候，在商用并行工程中，我们会使用守候线程技术，即针对一个端口，一个数据连接，我们不知道对方会什么时候传请求信令过来，因此，安排一个线程，不断循环检查，如果有请求信令到来，即接收信令并处理。

在数据传输类工程应用中，我们很注重业务层和传输层的分层概念，即底层传输层不理解业务数据本身，收到数据也不自己处理，而是通过回调函数，将数据传递给上层业务层处理。这是常见的传输手法。

但这时候就往往会出现一个问题，回调函数只传递一根参数指针不够用。我们要传递的参数，多于一个。对于这个问题，笔者通常建议采用参数结构体来封装。如下例：

```

//这是 Listen 模块线程回调函数的参数结构体
typedef struct _CListen_ListenAcceptTask_Param_
{
    Linux_Win_SOCKET m_nSocket;    //accept 到的 socket
    CListen* m_pThis;              //本对象指针（代 this）
}SCListenAcceptTaskParam;
//习惯写法，定义结构体尺寸，帮助动态内存申请
const ULONG SCListenAcceptTaskParamSize=sizeof(SCListenAcceptTaskParam);
//这是线程回调函数，静态函数
bool CListen::ListenAcceptTask(void* pCallParam,int& nStatus)
{
    //注意这个强制指针类型转换，指针类型用的是参数结构体类型，不是本类 CListen 类型
    SCListenAcceptTaskParam* pParam=
        (SCListenAcceptTaskParam*)pCallParam;
    //出于统一考虑，用 pThis 来调用
    CListen* pThis=pParam->m_pThis;
    //获得关键参数，socket
    Linux_Win_SOCKET s=pParam->m_nSocket;
    //...
    //这属于线程启动的远堆传参，参数结构是 malloc 出来的数据区，因此需要 free
    free(pCallParam);
}
//这是监听线程的启动示例
bool CListen::ListenTask(void)
{
    //...
}

```

```

s=accept(pThis->p.fd,          //当有连接请求到来, accept 得到新的 socket
        (struct sockaddr *)&accept_addr, (socklen_t*)&len);
//...
//注意此处, 远堆传参启动线程, 此处申请, 线程中释放
SCListenAcceptTaskParam* pParam=(SCListenAcceptTaskParam*)
    malloc(SCListenAcceptTaskParamSize);
if(pParam)
{
    pParam->m_nSocket=s;      //填入关键参数 socket
    pParam->m_pThis=this;    //关键点, 本对象指针 this 由此传入
    //这是启动线程函数, 后文线程池有专门讲述,
    //读者此处仅需关心启动了 ListenAcceptTask 县城,
    //并使用 pParam 作为参数指针即可
    StartTask(ListenAcceptTask, pParam);
}
//请注意, pParam 没有释放, 会在线程中释放。
//...
}

```

这个例子中, 请大家关注几点:

1、我们有一个线程回调函数 ListenAcceptTask, 它需要传输两个指针, 接收到的 socket 和本类的对象指针 pThis。

2、我们的线程池回调模型在设计时, 一般只透传一根 void* 指针, 因此, 我们使用了一个结构体来封装这两个参数。

3、由于是启动线程, 因此需要远堆传参, 这个结构体是启动函数动态在远堆中 malloc 申请的, 因此, 线程函数返回时, 需要自行释放, 这个地方不符合“谁申请、谁释放”原则, 请各位读者关注。

以上算是回调函数参数传递, 以及线程参数传递的常见收发, 这么书写程序, 虽然有点繁琐, 但方方面面的禁忌和顾虑都考虑到了, 从来不会出错, 因此, 建议各位读者多加参考。

3.11.3 事件模型和回调模型

这个其实不完全算是 C 和 C++ 无错化程序设计的范畴, 属于回调模型的一个更深层次的思考。

我们从上例可以看到, 上例我们监听模块在接收到一个新的 socket 连接后, 会启动一个线程来处理, 然后腾出自己的时间, 继续处理下一个监听请求, 以保证高吞吐率。这种“守候-并行处理”的模型在实际工程中应用很多, 最著名的, 就是 Windows 的事件处理体系。

学过 Windows 编程的, 特别是学习过 WinMain 方式, 以纯 C 开发 Windows 程序的读者可能知道, Windows 程序一旦启动, 在简单创建窗口后, 一般就进入一个死循环, 开始不断检测事件, 并及时处理。

这套机制就是 Windows 的事件通知体制, 在很古老的 DOS 时代, 操作系统并没有提供键盘、鼠标这类消息的自动读取服务, 需要每个应用程序自己书写读 IO 端口的代码, 这给程序设计带来很多不便。因此, Windows 系统从一开始, 就准备了一个全系统的标准事件检索服务, 检索包括鼠标、键盘、外设等各种输入设备的信息。

这个服务一旦检测到输入, 就会把相信的输入信息编辑成事件, 根据当前应用程序的关系, 查询到目标窗口, 并将该事件放入该窗口的事件队列, 然后继续检测。

而如前文，窗口应用程序的事件检查死循环，一旦从事件队列中检索到新的事件，就会处理。Windows 以这种异步机制，实现了多个应用程序并行，共享输入设备的工作模式。这套模式，通常我们称之为事件模型。

由此我们可以看到，其实这类对一些状态，一些输入进行守候跟踪检测的设计需求，其实有两种解决方案，一种是 Windows 的事件模型，另外一种是本书中讲到的回调模型。前者的特点是异步，各个模块工作和谐，并行的锁争用较少，系统比较安全，而后者回调模型的特点，是同步操作，速度快，效率高，但没有统一的机制，每种情况需要分别处理，锁的争用较高，如果不做谨慎的设计，很容易程序崩溃。

不过，由于商用并行工程，很多情况下的设计都讲究高效率，高并发性和高吞吐量，因此，笔者的工程中，回调模型使用较多，几乎不使用低速的事件队列模型。但这要求程序员对 C 语言的回调机制，设计方式，设计者和使用者的区别，线程的参数传递，锁争用机制等一系列知识点掌握牢固，才能设计出合用、安全的商用并行工程。

3.12 C 语言字符串的深入研究

在笔者刚刚开始使用 C 语言开发程序时，内存的读写出界是一个非常严重的问题，在 Windows 下，很可能导致蓝屏错误，而 Linux 下，则是直接引发段错误，导致程序崩溃。因此，在研究无错化程序设计方法时，笔者把这个问题作为重点问题来研究。

内存读写出界，顾名思义，是常出现于内存拷贝动作中的一种错误，我们知道，C 和 C++ 语言的内存拷贝，通常分为二进制拷贝 `memcpy` 和文本拷贝 `strcpy`。

但经过笔者研究，发现二进制拷贝时，发生错误的概率其实并不高，原因很简单，`memcpy` 要求程序员显式输入拷贝的字节数，由于程序员开发过程中，内存缓冲区一般都是自己申请的数组，因此一般这个拷贝长度都能正确输入，很少出错。

但另一方面，几乎所有的内存读写出界，都和字符串拷贝有关，因为在 C 语言中，字符串是一个引申出来的概念，仅仅是通过文本内容后缀一个 ‘\0’ 来描述，这就带来下面几个问题，引发了笔者的思考。

1、拷贝时，长度通常不需要程序员输入，而是隐含使用 ‘\0’ 作为结束，这样，拷贝的实际长度是隐含的，不直观，容易出错。

2、这个 ‘\0’ 字符本身是否计入拷贝的范围，是一个问题，很多时候，连程序员自己也是模糊的。

3、拷贝以来源字符串的 ‘\0’ 为结束，那么目的缓冲区如果不够的话，可能会有 bug。

4、拷贝出来的字符串是否在各种情况下，均以 ‘\0’ 结尾，这是一个问题。

5、其他的字符串处理函数，如 `snprintf` 字符串构造函数，是否有类似问题。

3.12.1 字符串拷贝

字符串拷贝可以说是 C 语言所有处理字符串函数的基础，C++ 语言中也很常用，笔者在此仔细比对。

3.12.1.1 不安全的 `strcpy`

首先，笔者写了这样一个测试函数，程序逻辑很简单，就是把一个字符串拷贝到另外一个字符串数组空间。

```
void strcpyTest0()
{
    int i;
    char szBuf[128];
```

```

    for(i=0;i<128;i++) szBuf[i]='*';
    szBuf[127]='\0';    //构造一个全部是*的字符串
    char szBuf2[256];
    for(i=0;i<256;i++) szBuf2[i]='#';
    szBuf2[255]='\0';    //构造一个全部是#的字符串
    strcpy(szBuf,szBuf2);
    printf("%s\n",szBuf);
}

```

但是，大家可能已经注意到，由于源字符串比目标地址要长，出现了内存写出界，因此，程序很悲惨地死去了。

3.12.1.2 还是不安全的 strcpy

通过上例，笔者发现自己需要在拷贝时多输入一个参数，来标明目的地址有多长，检查 C 语言的库函数说明，有一个 `strncpy` 可以达到这个目的，这个函数的原型如下：

```
char *strncpy( char *strDest, const char *strSource, size_t count );
```

好了，看起来这次我们的问题解决了，于是笔者写出了如下代码：

```

void strcpyTest1()
{
    int i;
    char szBuf[128];
    for(i=0;i<128;i++) szBuf[i]='*';
    szBuf[127]='\0';
    char szBuf2[256];
    for(i=0;i<256;i++) szBuf2[i]='#';
    szBuf2[255]='\0';
    strncpy(szBuf,szBuf2,128);    //注意，这里换了拷贝函数
    printf("%s\n",szBuf);
}

```

一切都显得很好，但是，当笔者检查输出结果的时候，发现了问题，字符串后面有时会跟几个奇怪的字符，好像没有用“\0”结束，于是笔者把上面的拷贝语句改成

“`strncpy(szBuf,szBuf2,8);`”，只拷贝 8 个字符，问题出现了，程序输出如下：

```

#####
*****

```

果然，当请求的目标地址空间比源字符串空间要小的时候，`strncpy` 将不再用“\0”来结束字符串。这是一个巨大的隐患，因为虽然本次拷贝一定会成功，但本次拷贝的结果不再是 C 语言合法的字符串，下次针对其的任何操作将会失败，最可怕的是，出现 bug 的地方离这个问题点很远，bug 基本不可跟踪。

3.12.1.3 安全地字符串拷贝函数

笔者仔细想了想，认为自己需要如下一个字符串拷贝函数：

- 1、允许用一个整数界定目标地址空间尺寸。
- 2、当目标地址空间 `nD` 小于源字符串长度 `nS` 时，应该只拷贝 `nD` 个字节。
- 3、任何情况下，目标地址空间均应该以“\0”结束，保持一个合法的字符串身份。因此，得到的字符串最大长度为 `nD-1`。

于是，笔者再次写了一个新的字符串拷贝函数：

```

void xg_strncpy1(char *pD, char *pS,int nDestSize) //必须输入目标空间大小
{
    memcpy(pD,pS,nDestSize);    //强行以二进制定长拷贝
    *(pD+nDestSize-1)='\0';    //强行进行“\0”修饰
}

```

```
}
```

很容易是不？将这个拷贝函数代入前文的例子，只输出 7 个`#`，结果完全正确。

3.12.1.4 内存读错误的思考

本来以为可以就此打住了，不过，没多久，笔者就发现一个奇怪的现象，上述函数在 VC6.0 的 Debug 模式下有错误，但是 Release 模式下却一切正常。由于笔者的工程都是以 Release 发布，因此，这个现象影响并不大。

但笔者奇怪了很久，终于忍不住想看看这个问题到底是怎么回事。笔者把上面的 memcpy 用自己的一个复制循环代替，单步跟踪，想看看究竟怎么回事？

原因找到了，笔者的测试代码，希望拷贝一个字符串，到一个 256 字节长的数组空间，但是，拷贝到第 33 字节时出错，原因很简单，源字符串空间只有 32 Bytes，出现了内存读错误。

原来，笔者上面的代码只是防止了内存写出界，但没有针对读出界进行检查，在 VC 的 Debug 模式下，内存读出界也是一种非法错误，因此被报错。

知道了原因，解决就很简单了，我把上面的拷贝函数改成如下写法，经测试，一切正常，再没有出过问题。

```
void xg_strncpy2(char *pD, char *pS,int nDestSize)
{
    int nLen=strlen(pS)+1;
    if(nLen>nDestSize) nLen=nDestSize;        //目标地址和拷贝地址取小
    memcpy(pD,pS,nLen);
    *(pD+nLen-1)='\0';
}
```

3.12.2 字符串构造

字符串构造是 C 语言字符串函数族的另外一个重要功能，从最简单的屏幕输出，到复杂的网络文本信令构建，字符串构造函数均起着重要的作用，在我们商用工程中更是频繁使用。因此，在做完拷贝的验证后，笔者继续验证了字符串构造函数。

3.12.2.1 不安全的 sprintf

笔者尝试着写了如下一个函数，经过测试，果然也发生了崩溃。

```
void sprintfTest0()
{
    int i;
    char szBuf[128];
    for(i=0;i<128;i++) szBuf[i]='*';
    szBuf[127]='\0';
    char szBuf2[256];
    for(i=0;i<256;i++) szBuf2[i]='#';
    szBuf2[255]='\0';
    sprintf(szBuf,szBuf2); //由于来源>目的，拷贝造成内存写错误，崩溃
    printf("%s\n",szBuf);
}
```

3.12.2.2 还是不安全的_sprintf

查阅库函数手册，笔者找到这么一个函数_sprintf，其函数原型如下：

```
int _sprintf( char *buffer, size_t count, const char *format [, argument] ... );
```


这个函数允许界定目标地址尺寸，但是，由于研究拷贝函数的经验，笔者怀疑它也有 `strncpy` 相同的问题，因此，写了这么一段代码测试：

```
void sprintfTest1()
{
    int i;
    char szBuf[128];
    for(i=0;i<128;i++) szBuf[i]='*';
    szBuf[127]='\0';
    char szBuf2[256];
    for(i=0;i<256;i++) szBuf2[i]='#';
    szBuf2[255]='\0';
    _snprintf(szBuf,8,szBuf2); //只打印 8 个字符，看有没有正常的“\0”结尾
    printf("%s\n",szBuf);
}
```

果然，程序输出如下同样的错误，没有用“\0”结束。

```
#####
*****
```

另外，笔者还发现了另外一个不足，就是当出现异常时，`_snprintf` 函数返回-1，不再返回打印的字符数，那么使用如下代码将会造成逻辑错误，甚至可能崩溃：

```
char szBuf[256];
int nCount=0;
while(1) //这里表示循环构造
{
    nCount+=_snprintf(szBuf+nCount,256-nCount,"... ");
    //多个字符串构造成一个字符串，需依赖 snprintf 返回的长度
}
```

注意，代码利用 `_snprintf` 返回的值，来确定下一个起始点，这在 C 语言中是很常用的手法，但是，当 `_snprintf` 返回-1 的时候，有可能会写到*(szBuf-1)的位置上，典型的内存写界错误。

3.12.2.3 安全的字符串构造函数

经过仔细思考，笔者构造了一个安全的字符串构造函数：

```
int xg_printf(char* szBuf,int nDestSize,char *szFormat, ...)
{
    int nListCount=0; //计数器
    va_list pArgList;
    va_start (pArgList,szFormat); //变参处理循环
    nListCount+=_vsprintf(szBuf+nListCount, //处理每一个变参参数
        nDestSize-nListCount,szFormat,pArgList);
    va_end(pArgList);
    *(szBuf+nDestSize-1)='\0'; //人工补充'\0'
    return strlen(szBuf); //返回真实的字符串长度
}
```

注意，这里笔者采用了变参函数设计，为的是调用模型和 `sprintf` 一样方便，以便无缝切换。另外，最后一个 `return` 也非常重要，因为很多场合，我们需要知道究竟打印了多少字符。将这段函数代入前文的例子后一切正常。

3.12.3 关于字符串处理的结论

笔者在本小节，用了大量的篇幅，详述 C 语言字符串函数的实验细节和结论，主要的原因就是这类字符串相关操作，在商用工程中非常重要，而且也是 C 和 C++ 无错化程序设计的重要组成部分。

如果各位读者不能仔细分辨其细微的差异性，则很可能在未来的工程开发中，犯下无数的 bug，并且，为了解决这些 bug，会耗费大量的时间。因此，建议各位读者仔细阅读和体会。

上述试验是笔者在 2000 年左右完成的，当笔者完成上述安全代码之后，这么多年，笔者的程序再没有出过内存读写出界的 bug，这使得笔者的工程代码具有很高的安全性。

另外还有一个额外的好处，笔者做的商用服务器，由于大量采用了上述函数来完成字符串的拷贝和构造工作，事实上对每次拷贝行为都有严格的目标边界界定，从客观上，杜绝了缓冲区溢出攻击的可能性，这使得笔者的商用工程，不必像微软的 Windows 系列一样，不断的做补丁修补工作。

3.13 C/C++ 语言无错化程序设计小结

在本章中，笔者就个人的经验，向大家论述了很多关于无错化程序设计的方法，但笔者认为这些并不是重点。

无错化程序设计是一个很大的话题，不仅仅本章所简述的内容，在以后的章节中，大家在内存池的设计，在线程池、任务池的构建，以及工程库的规划本身等各个方面，可以看到大量无错化程序设计的原则的实战应用。

在前文中，大家可能看到，笔者为了实现无错化设计，动用了很多方法，从最简单的习惯养成，到项目组的工程规范等等，目的只有一个：“**减少 bug，提高程序员的生产率**”。

但这些并不重要，笔者在此想和各位读者分享的重点是：“**思考并设计这些方法的原则，动机，才是实现无错化程序设计的关键**”。

因此，笔者建议大家以后的工作中，可以慢慢培养如下的思路 and 原则，主动去发现问题和解决问题，才能一步步走向成功商用程序员之路。

- 1、善于总结和分析自己的程序，对 bug 和常见故障点，进行自我归类总结。
- 2、对于频繁发生的 bug，做出仔细的研究，研究其成因，特别是思考自己有什么思维习惯会导致这类 bug，即找出 bug 的思想出发点。
- 3、对于容易导致 bug 之处，自己总结成熟的工程代码替换，尽量使一段代码，解决一方面的问题。
- 4、对于代码可能不好控制的 bug，强迫自己养成习惯，以一个好的习惯，解决整整一个方面的问题。这包括书写习惯，输入习惯，函数和类的设计习惯。
- 5、对于新出现的需求，没有接触过的需求，习惯以无错化程序设计的思维进行思考，主动考虑可能的故障点，并及时作出预防，对于某些模糊不清之处，要习惯做实验确认。
- 6、无错化程序设计的关键，不是杜绝 bug，而是有一套行之有效的方法，使 bug 尽早暴露出来，便于分析和跟踪，因此养成 log 习惯，在程序中预设方便的 debug 功能，将是无错化设计的关键。
- 7、严格遵守公司以及项目团队的开发规范，这些大多是前人总结的成果，很多都能帮助我们有效避免 bug，建议各位读者首先是尊重，再谈改进。

- 第 4 章 设计自己的工程库
 - 4.1 数据传输库中到底需要哪些模块
 - 4.1.1 跨平台定义
 - 4.1.2 锁与安全模块
 - 4.1.3 内存池
 - 4.1.4 资源管理池
 - 4.1.5 线程池与任务池
 - 4.1.7 队列管理
 - 4.1.8 其他工具
 - 4.2 工程库基础---跨平台定义
 - 4.2.1 锁定义
 - 4.2.2 线程控制相关定义
 - 4.2.3 Socket 传输相关定义
 - 4.2.4 include 系统头文件

第 4 章 设计自己的工程库

笔者曾经分析了一下各种语言的进步历程，发现一个很有趣的现象。如 VC++ 的各个版本进步，除了个别的 bug 消除，做出一些补丁之外，绝大多数是对内置各种工程库的添加和维护。其他几门语言，如 C++Builder、Dephi 等，情况也差不多。

在程序界，笔者也遇到了无数次讨论，特别是年轻的大学生，热衷于比较各个语言产品的优劣性，在这些讨论中，笔者关注到最多的论据，就是某个语言拥有某个特性，可以很方便地实现某个功能这类的提法。这使得笔者常常有种哭笑不得的感觉。

其实就笔者看来，这类讨论毫无意义。比如 VC++ 和 C++ Builder，二者既然以 C++ 为名，则必然要遵守 C++ 的标准和规范，可能不一定 100%，但至少常规的 C++ 程序书写和编译应该通用。二者所差别的，其实是内部包含的工程开发库的差异，VC++ 提供 MFC 类库，而 C++ Builder 提供 OWL 类库。二者其实都是对 Win32API 的封装，从本质上没有差别。仅仅是开发者的理解和理念不同罢了。

不过这说明一个道理，其实语言的发展，就是工程库的发展，微软等软件公司总结了语言使用过程中，比较有代表性的需求，常用写法，整理出较为通用和标准的一些程序模块，在语言中提供给程序员使用，仅此而已。

但这就带来一个问题，作为通用的编程语言，其工程库的设计，很难预测所有的开发需求，因此必然采用很多通用的，多重考虑的设计方案，其效率偏低，不能满足使用要求。

比如在游戏界，性能优化是重点，通用库就很难满足其性能要求，这体现在 Windows 下，游戏开发很难利用 GUI 这类通用库来完成，微软被迫单独开发了 DirectX 来实现高速游戏开发。

在我们的商用工程项目中，很多时候也会面临对某个性能的极限优化需求，如某台服务器的连接数做到最大，吞吐量做到最大等，这些，也都很难使用通用库来完成。因此，笔者建议，在商用工程领域，程序员应该主动针对自己常见的工程需求，积累一些自己的工程库。

工程库还有一个额外的好处，入库的代码通常是经过前面工程长期测试的结果，因此，bug 很少或者没有，不断积累和重用工程库，可以使新开发的工程，bug 率越来越低，最终实现无错化程序设计。

提示：虽然前文笔者详细向大家介绍了 C 和 C++ 语言无错化程序设计的方法和原则，但在后文介绍的工程库代码中，各位读者可能仍然能看到某些不太符合这些原则之处。这个原因很简单，很多代码是笔者产生无错化思想之前的产物，或者说，无错化思想本身就是伴随工程库的建立而成熟的，这样反过来看前期的代码，自然有很多不够科学之处。

不过，这些代码，很多历史已经和久远，在很多商用工程中得到使用和验证，因此，笔者最终决定不要做任何修改，避免引入新的 bug。

这也体现出工程库思想的一个重要原则，尊重永远比创新和规范重要，一段代码，只要有简明的调用接口，并通过实践验证为稳定代码，则尽量不要修改，尤其不要为修改而修改。

4.1 数据传输库中到底需要哪些模块

通常我们很难界定一个工程库到底需要包含哪些内容，不过就笔者一直从事商用并行开发，因此自己的工程库需求相对单一，这里可以简要列出下列必要元素。当然，各位读者根据自身的开发需求，还可能进一步增补。

4.1.1 跨平台定义

前文所述，商用并行工程，一般都有跨平台开发需求，而为了降低程序员的工作量，针对同一逻辑维护多套代码是不经济，也不科学的，因此，工程库首先的需求就是跨平台定义，即同一工程库可以在多个平台下编译运行。

不过，通常 UI 设计很难做到跨平台开发，就 C 和 C++ 语言而言，Windows 下的窗口编程和 Linux 的 Xwindows 开发有很大差异性，代码基本不可能通用，因此，跨平台的工程库，很少包含 UI 代码，一般仅包含数据传输、内存管理、时间片管理相关内容。这要求系统设计时，将 UI 与业务层分离，UI 层可以调用工程库代码，但工程库代码不能依赖 UI 相关内容，以免出现跨平台干涉。

同理，跨平台的工程库，很难依赖某种语言特性，比如一个链表类，我们如果使用了 MFC 提供的 CList 定义，则很难移植到 Linux 下，因此，跨平台工程库，很多基本的数据结构需要程序员自行开发。

借助 STL 也许是个不错的主意，不过，大家可能没有关注到，很多嵌入式平台的交叉编译器，考虑到嵌入式平台内存偏小，资源不多的特点，很可能不提供 STL 功能，因此，嵌入式程序员要小心。

另外，以笔者经验，STL 等通用库的设计时，通常会以 PC 机作为目标运行平台，因此，其算法设计对于内存的边界考虑较少，优化方向，也大多采用“空间换时间”的原则，这在某些只有 64M RAM 的嵌入式平台上，会成为很沉重的负担，甚至不可使用。

因此，跨平台工程库的开发和维护，对程序员有更高的要求，需要程序员对于很多基础知识的掌握非常透彻，并经过长期实践和测试，方能完成。

4.1.2 锁与安全模块

一般说来，商用并行工程，对于各种锁的定义，以及各种安全模块（如前文所述的字符串安全处理函数），是基本配置。

各个操作系统，都提供了不止一种锁的定义，如 Windows 的临界区，Linux 下的原子锁，以及大家都有的信号量等。但我们知道，复杂的定义会直接带来工程库的复杂度，导

致维护不变，因此，本书将所有线程安全访问相关保护机制，均归纳为“锁”，并实现统一的接口进行管理。

4.1.3 内存池

这在前文已经略有涉猎，不过此处还是要强调一下。几乎所有的 C 和 C++ 程序员都知道，操作系统已经为我们提供了内存管理功能，我们可以在运行期自由地进行内存的动态申请和释放，对象的创建和摧毁。这是 C 和 C++ 语言巨大的优势体现。

但很不幸，各个操作系统，均是通用操作系统，即一个平台，要为多种语言的程序运行服务，因此，无论是 Windows 还是 Linux，均没有针对 C 和 C++ 语言做特定的内存管理优化。

这表现到具体程序上，就是内存泄漏与否，请各个应用程序自行检查，操作系统不关心内存申请的合理性，仅仅根据资源情况返回分配结果，即，操作系统不会主动帮助 C 和 C++ 程序员检查内存错误。

前文已经提过，即使我们解决了所有内存泄漏问题，仅仅是内存碎片，也可能导致一个 7*24 小时的服务程序在将来某个时刻挂死。成为 bug。因此，C 和 C++ 语言的商用并行工程库，必须重点考虑内存的可重用问题，因此，构建自己的内存池势在必行。本书也将重点讨论这一话题。

4.1.4 资源管理池

这是笔者的经验之谈，在思考无错化程序设计方法时，笔者发现有很多二元动作，如果不小心控制，会导致 bug。进一步分析发现，这类二元动作，一般都与资源的申请与释放相关，如内存资源，socket 资源，锁资源等。

因此，笔者就思考，如果能建立一套机制，强制在资源申请和释放的时候，实现一套注册和反注册机制，并在程序退出时打印出泄漏部分，那么，对于由于程序员遗忘而导致的资源泄漏问题，二元动作不规范问题，就可以很轻松的解决。

因此，笔者在内存池的基础上，进一步建立了资源管理池概念，代程序员管理各种资源的申请和释放，并实现报警提示机制，经过使用效果很好，从 2001 年之后，笔者和笔者的项目团队，基本没有出现资源泄漏情况。本书也将重点介绍这一方法。

4.1.5 线程池与任务池

作为以并行处理为核心的商用工程，线程几乎是所有程序设计的核心，非常重要。但遗憾的是，并行开发往往属于操作系统特性，作为纯粹的语言教学，C 和 C++ 的相关教材讨论得较少，而操作系统的开发书籍，又偏重于本操作系统的特性，很少针对某个特定语言来详细论述。

这无形中形成了一个空区，一个程序员，很难通过市场上的书籍，掌握并行计算的精髓，也很难写出合格的并行计算程序。

本书出于讨论商用并行工程的目的，有必要详细讨论并行计算的方方面面，因此，本书后文的工程库，将向大家展示线程池的设计方法，并在此基础上，进一步拓展为任务池这一通用并行计算模型，最终使大家获得一个高可用性的并行计算基本库。

4.1.7 队列管理

在笔者这么多年的工作经验中，发现各种经典数据结构中，队列是商用并行工程最常用的一种。无论是同步和异步动作的切换，还是传输之间的缓冲，以及跨线程、跨进程通信，队列几乎无处不在。

因此，本书提供的工程库，把队列作为重点向大家介绍，并详细列举了常见的几种常见的队列模型，以及其特征使用情况，供大家参考。

4.1.8 其他工具

当然，商用并行工程，作为商业性软件工程的一种，除了其关心的核心业务，一些相关的辅助模块也是非常必要的，如 log 日志模块，Debug 模块等，这些模块在构建高稳定性、高可用工程模块中至关重要，本书也会做详细介绍。

4.2 工程库基础----跨平台定义

作为后续具体工具介绍的基础，我们有必要先预定义一些跨平台相关的定义，根据笔者经验，商用数据传输工程中，常见跨平台差异最大，需要利用编译宏预定义的，有如下几类功能调用。

4.2.1 锁定义

4.2.1.1 基本跨平台锁定义

锁是一切并行计算的基础，并且，锁属于操作系统特性，与 OS 密切相关，在 Windows 下和 Linux 下，锁的使用有很多差异性，为了简化程序开发难度，便于工程库管理，这里笔者给出一个比较通用的定义。

```

#ifdef WIN32    //Windows 下定义
//定义锁变量类型
    #define MUTEX CRITICAL_SECTION
//定义初始化锁的功能
    #define MUTEXINIT(m) InitializeCriticalSection(m)
//定义加锁
    #define MUTEXLOCK(m) EnterCriticalSection(m)
//定义解锁
    #define MUTEXUNLOCK(m) LeaveCriticalSection(m)
//定义摧毁锁变量操作
    #define MUTEXDESTROY(m) DeleteCriticalSection(m)
#else //Linux 下定义
//定义锁变量类型
    #define MUTEX pthread_mutex_t
//定义初始化锁的功能
    #define MUTEXINIT(m) pthread_mutex_init(m, NULL) //TODO: check error
//定义加锁
    #define MUTEXLOCK(m) pthread_mutex_lock(m)
//定义解锁
    #define MUTEXUNLOCK(m) pthread_mutex_unlock(m)
//定义摧毁锁变量操作
    #define MUTEXDESTROY(m) pthread_mutex_destroy(m)
#endif

```

4.2.1.2 C++语言的锁对象

当然，我们这里给出的是纯 C 语言的调用模型，不过很多时候，由于 C++面向对象的方便性，可能锁的使用者不希望去关心锁初始化，摧毁之类的细节，仅仅希望简单的提供 Lock 和 Unlock 两个接口即可。因此，这里笔者也提供了一个基本的锁类，方便调用。

```

class CMutexLock
{
public:
    CMutexLock(void) {MUTEXINIT(&m_Lock);}           //构造函数，初始化锁
    ~CMutexLock(void) {MUTEXDESTROY(&m_Lock);}        //析构函数，摧毁锁
public:
    void Lock() {MUTEXLOCK(&m_Lock);}                 //加锁操作
    void Unlock() {MUTEXUNLOCK(&m_Lock);}              //解锁操作
private:
    MUTEX m_Lock;                                     //锁变量（私有）
};

```

大家可能注意到，这个锁类非常简单，并且全部使用 C++的内联模式写法，保证高效性。这可是说是笔者所有并行代码中，锁的根基，所有的锁应用，均由此引申。

这里提示大家一点，很多时候，功能强大的类，其实可以写得很简单，

4.2.2 线程控制相关定义

线程的操作与操作系统是密切相关的，可以说，跨平台差异性最大的就在这一部分，这里笔者给出一个常用的定义。

```

#ifdef WIN32    //Windows 下定义
    #include <process.h>
//线程句柄类型
    #define THREAD HANDLE
//线程 ID 类型
    #define THREADID unsigned
//线程启动函数统一构型，注意函数型宏的使用
    #define THREADCREATE(func, args, thread, id) \
        thread = (HANDLE)_beginthreadex(NULL, 0, func, (PVOID)args, 0, &id);
//线程启动失败后返回错误码定义
    #define THREADCREATE_ERROR NULL
//线程函数标准构型
    #define THREADFUNCDECL(func,args) unsigned __stdcall func(PVOID args)
//工程中通常需要检测本次开机以来的毫秒数，Windows 和 Linux 不一样，此处予以统一。
    #define _GetTimeOfDay GetTickCount
//Windows 下最小睡眠精度，经实测为 10ms
    #define MIN_SLEEP 10
#else //Linux 下定义
//线程句柄类型
    #define THREAD pthread_t
//线程 ID 类型
    #define THREADID unsigned //unused for Posix Threads
//线程启动函数统一构型，注意函数型宏的使用
    #define THREADCREATE(func, args, thread, id) \
        pthread_create(&thread, NULL, func, args);
//线程启动失败后返回错误码定义
    #define THREADCREATE_ERROR -1
//线程函数标准构型
    #define THREADFUNCDECL(func,args) void * func(void *args)
    // #define Sleep(ms) usleep((__useconds_t)(ms*1000))
//工程中通常需要检测本次开机以来的毫秒数，Windows 和 Linux 不一样，此处予以统一。
    unsigned long GetTickCount(void);
    #include <sys/time.h>
    #define _GetTimeOfDay GetTickCount
//Linux 下没有 Sleep 函数，为了便于统一编程，仿照 Windows 下定义该函数
    #define Sleep(ms) usleep(ms*1000)
    #define MIN_SLEEP 1
#endif

```

特别值得一提的是 `GetTimeOfDay` 这个函数，这是函数的功能检测开机以来的毫秒数，在很多构建 ID，精确计时的场合，非常有用，但很可惜，VC 下和 gcc 下由于操作系统的差异，这个函数的实现差别很大，很多程序员都不知道怎么跨平台调用。这里笔者给出一个通用写法。


```

#ifdef WIN32    //Windows 下定义
//Windows 下有此函数，无需重复定义
#else    //Linux 下定义
//获得本次开机以来毫秒数
Inline unsigned long GetTickCount(void)
{
    unsigned long lRet=0;
    struct timeval tv;
    gettimeofday(&tv,null);
    lRet=tv.tv_usec/1000;
    return lRet;
}
#endif

```

4.2.3 Socket 传输相关定义

Socket 传输函数族在 Windows 和 Linux 下，虽然都号称支持伯克利标准，但由于系统先天的差异性，二者差别比较大，需要做出专门的适应性定义。经过笔者分析，socket 函数族的跨平台差异性主要体现在如下几个方面：

- 1、Windows 下以 unsigned int 作为 socket 套接字的变量类型，而 Linux 下为 int，这直接体现在当一个 socket 创建错误后，标示错误的 socket 代码，以及错误码，虽然都为 0xFFFFFFFF，但 Linux 下是 -1，而 Windows 下是一个极大的正数。因此，Linux 下常见的写法 if(0>nSocket)，在 windows 下是明显错误的。

- 2、某些特定的函数，setsockopt、getsockopt、send、recv，在某个参数的定义上，Windows 和 Linux 下差别较大，不是一个变量类型，必须予以关注。

- 3、Windows 的各种错误定义不符合伯克利标准，尤其是各种错误的常量名，也和标准不一样，必须予以统一，否则程序不能实现跨平台编译。

- 4、Windows 下使用 Socket 函数族需要特殊的初始化函数和结束函数，这和 Linux 明显不同，需要在程序设计中特别关注。

综上所述，Socket 函数族的访问，在跨平台整合上比较困难。笔者为此也很困惑，最后，笔者决定，将所有的跨平台宏定义，向伯克利标准靠拢，以 Linux 的为准。这里，笔者给出我们工程中常见的标准。

```

#ifdef WIN32    //Windows 下定义
    #include <winsock.h>
//定义 Socket 套接字变量类型
    #define Linux_Win_SOCKET SOCKET
//标准的 Socket 关闭函数，不过，由于后文资源池接管了关闭动作，此处隐去定义
    // #define Linux_Win_CloseSocket closesocket
//协议族命名约定
    #define Linux_Win_F_INET AF_INET
//非法的 socket 表示值定义
    #define Linux_Win_InvalidSocket INVALID_SOCKET
//标准 socket 错误返回码
    #define Linux_Win_SocketError SOCKET_ERROR
//setsockopt 第 4 个变量类型定义
    #define Linux_Win_SetSockOptArg4UseType const char
//getsockopt 第 4 个变量类型定义
    #define Linux_Win_GetSockOptArg4UseType char
//send recv 函数的最后一个参数类型
    #define Linux_Win_SendRecvLastArg 0
//注意此处，所有的错误返回码定名，Windows 平台向标准的伯克利 socket 规范靠拢。
    #define EWOULDBLOCK          WSAEWOULDBLOCK //10035
    #define EINPROGRESS          WSAEINPROGRESS
    #define EALREADY             WSAEALREADY
    #define ENOTSOCK             WSAENOTSOCK
    #define EDESTADDRREQ         WSAEDESTADDRREQ
    #define EMSGSIZE             WSAEMSGSIZE
    #define EPROTOTYPE           WSAEPROTOTYPE
    #define ENOPROTOOPT          WSAENOPROTOOPT
    #define EPROTONOSUPPORT       WSAEPROTONOSUPPORT
    #define ESOCKTNOSUPPORT       WSAESOCKTNOSUPPORT
    #define EOPNOTSUPP           WSAEOPNOTSUPP
    #define EPFNOSUPPORT          WSAEPFNOSUPPORT
    #define EAFNOSUPPORT          WSAEAFNOSUPPORT
    #define EADDRINUSE           WSAEADDRINUSE
    #define EADDRNOTAVAIL        WSAEADDRNOTAVAIL
    #define ENETDOWN             WSAENETDOWN
    #define ENETUNREACH           WSAENETUNREACH
    #define ENETRESET            WSAENETRESET
    #define ECONNABORTED          WSAECONNABORTED //10053
    #define ECONNRESET           WSAECONNRESET //10054
    #define ENOBUFS              WSAENOBUFS
    #define EISCONN              WSAEISCONN
    #define ENOTCONN             WSAENOTCONN
    #define ESHUTDOWN            WSAESHUTDOWN
    #define ETOOMANYREFS         WSAETOOMANYREFS
    #define ETIMEDOUT            WSAETIMEDOUT
    #define ECONNREFUSED         WSAECONNREFUSED
    #define ELOOP                WSAELOOP
    #define EHOSTDOWN            WSAEHOSTDOWN
    #define EHOSTUNREACH         WSAEHOSTUNREACH
    #define EPROCLIM             WSAEPROCLIM
    #define EUSERS               WSAEUSERS
    #define EDQUOT               WSAEDQUOT
    #define ESTALE               WSAESTALE
    #define EREMOTE              WSAEREMOTE

```

```

#else    //Linux 下定义
//定义 Socket 套接字变量类型
    #define Linux_Win_SOCKET int
//标准的 Socket 关闭函数，不过，由于后文资源池接管了关闭动作，此处隐去定义
    // #define Linux_Win_CloseSocket close
//协议族命名约定
    #define Linux_Win_F_INET AF_INET
//非法的 socket 表示值定义
    #define Linux_Win_InvalidSocket -1
//标准 socket 错误返回码
    #define Linux_Win_SocketError -1
//setsockopt 第 4 个变量类型定义
    #define Linux_Win_SetSockOptArg4UseType void
//getsockopt 第 4 个变量类型定义
    #define Linux_Win_GetSockOptArg4UseType void
//send recv 函数的最后一个参数类型
    #define Linux_Win_SendRecvLastArg MSG_NOSIGNAL
#endif

```

另外，由于 Windows 在起停 socket 传输应用时，需要额外的初始化和结束代码，因此，请各位读者在开发独立的传输程序时，参考下列代码设计，在程序的开始和结束处添加相应代码。

```

#if defined(_MSC_VER)
    static WSADATA wsaData;          //中间 Socket 信息保存变量
    #pragma comment(lib,"wsock32") //dll 库调用准备
#else // not WIN32
#endif
void Linux_Win_Init()                //win 32 socket 初始化代码
{
#ifdef WIN32
    WORD wVersionRequested;
    int err;
    wVersionRequested = MAKEWORD( 2, 2 );
    err = WSAStartup( wVersionRequested, &wsaData );
    if ( err != 0 )
    {
        //输出错误信息给用户，提醒初始化失败
    }
#else // not WIN32
#endif
}
void Linux_Win_Exit()                //win32 socket 结束代码
{
#ifdef WIN32
    if ( LOBYTE( wsaData.wVersion ) != 2 ||
        HIBYTE( wsaData.wVersion ) != 2 )
    {
        //输出错误信息给用户，提醒版本不对
        WSACleanup( );
    }
#else // not WIN32
#endif
}

```

4.2.4 include 系统头文件

C 和 C++语言有一点比较麻烦，就是很多时候，需要 **include** 大量的系统头文件实现功能。很多时候，光是查找头文件，就浪费了程序员大量的时间。

笔者这里简要给大家提示一段工程库常用的头文件包含关系，只要按照这个包含关系，一般实现跨平台商用数据传输工程，已经够用了。

```
//通用包含
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <time.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <signal.h>
#ifdef WIN32 //windows 下特定包含
    #include <stdio.h>
    #include <conio.h>
    #include <windows.h>
    #include <process.h>
    #include <winsock.h>
#else //Linux Unix 下特定包含
    #include <unistd.h>
    #include <errno.h>
    #include <pthread.h>
    #include <fcntl.h>
    #include <unistd.h>
    #include <netinet/in.h>
    #include <string.h>
    #include <sys/time.h>
    #include <arpa/inet.h>
    #include <errno.h>
    #include <termios.h>
    #include <netdb.h>
    #include <getopt.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    #include <unistd.h>
#endif
#endif
```

第 5 章 Debug 工具

- 5.1 变参函数的设计
- 5.2 文本输出
 - 5.2.1 获得时间戳
 - 5.2.2 同时输出到文件和屏幕
 - 5.2.3 文本输出的原则
- 5.3 二进制输出的 Debug 函数
- 5.4 核心 Debug 和日志系统的区别
- 5.5 统计模块
 - 5.5.1 累加器
 - 5.5.2 Δ 计算模块
 - 5.5.3 平均值计算
 - 5.5.4 统计平均值计算
 - 5.5.5 辅助功能函数
- 5.6 CLowDebug 工具类
 - 5.6.1 需求分析
 - 5.6.2 数据边界声明
 - 5.6.3 类声明
 - 5.6.4 类工具函数
 - 5.6.5 业务函数
- 5.7 基本 Debug 工具小结

第 5 章 Debug 工具

在商用并行工程领域，很多时候我们的程序模块工作在后台，数据并无直观的显示界面，这给程序员 Debug，管理员的运维分析都带来了很大不便。因此，我们必须从一开始，就着手准备一套工具，帮助我们在以后的开发和运营中，能对各种运行数据进行实时观察和事后分析。

另一方面，在一个公网运行的服务系统上，数据的流动是随时发生的，对于一条数据信令而言，它从被生产者生产出来开始，或者存在于某一台服务器的收发队列里面，或者正在网络上被发送，或者已经被执行到数据库中，因此，很多时候网络数据信令的流动，只能被分析，无法被观察，对程序调试和运维监控造成了很大的困难。

这也是我们需要一套工具，来帮助我们跟踪、记录数据信令的流动，最终实现数据分析的目的。综上所述，我们在设计传输库的时候，首先需要设计 Debug 模块。下面，我们来分析一下商用数据传输系统 Debug 模块的需求：

- 1、在运行时，出现任何不可恢复性错误，系统趋于崩溃时，我们需要留下日志文件，以尽可能多的记录崩溃原因。
- 2、在运行时，如果出现任何临时性错误，系统自动处理并予以恢复，我们需要记录该事件的发生经过，处理策略，以及处理结果。
- 3、关键事件必须有记录，如重大模块、守候线程的起停，用户上下线时间以及原因等。
- 4、记录时，必须保留每条记录的时间戳，以便做时序分析。且每条记录均同时可以输出到屏幕和磁盘文件，并提供相应的开关，方便后续操作。
- 5、对于程序员或操作员关心的数据信令，应该能以文本和二进制两种模式实现显示和保存，方便观察。
- 6、对于程序员或操作员关心的系统性能指标，应该能提供出方便的统计模块，并能快速简便地统计出基本值，提供后续性能分析算法的使用。
- 7、Debug 模块作为核心库模块，必须保持高度稳定性和简便易用性，尽量使用静态数据结构，使用栈空间，不能有内存泄露，内存碎片等问题，必要时，牺牲功能，首先保证稳定性。或采用分层开发的方式，实现功能与性能的平衡。
- 8、作为跨平台的数据传输库的核心代码，Debug 模块的设计应该尽量减少对各个操作系统特性、方言的依赖，实现无缝移植。并且，Debug 模块必须做到多线程安全。

5.1 变参函数的设计

根据笔者的经验，一个工具模块，首先的要求就是好用，不好用的工具，不如没有。而 Debug 模块，由于涉及到大量的输出工作，因此，如何方便地格式化输出数据，就变成 Debug 模块首先要解决的问题。

所幸，C 语言已经给了我们答案，所有的 C 程序员，从第一天写出自己的“Hello World!”程序开始，就已经开始接触 C 语言的标准格式化输出，这套格式，可以说已经成为 C 语言世界的通用语言了。我们完全可以照搬过来，作为我们的格式化输出描述方式。

我们要做的无非是，仿照 C 语言 `printf` 函数的构型，实现一个变参格式化输出函数，并把用户希望输出的信息，输出到指定的字符串缓冲区，供后续操作写入磁盘或者打印到屏幕而已。

在前文中我们专门研究了字符串的安全构造和拷贝函数，因此，在我们的工程库中，统一使用这种模型来处理字符串，这里是一个基本的变参输出函数

```

//安全的变参输出函数
//szBuf: 用户指定的输出缓冲区
//nMaxLength: 用户指定的输出缓冲区尺寸
//szFormat: 格式化输出字符串（变参，可以是多个）
//返回输出的字符总数（strlen 的长度，不包括最后的'\0'）
int SafePrintf(char* szBuf,int nMaxLength,char *szFormat, ...)
{
    int nListCount=0;
    va_list pArgList;
    //此处做安全性防护，防止用户输入非法的缓冲区，导致程序在此崩溃。
    if (!szBuf) goto SafePrintf_END_PROCESS;
    //此处开启系统循环，解析每条格式化输出字符串
    va_start (pArgList,szFormat);
    nListCount+=Linux_Win_vsnprintf(szBuf+nListCount,
        nMaxLength-nListCount,szFormat,pArgList);
    va_end(pArgList);
    //实现缓冲区超限保护
    if(nListCount>(nMaxLength-1)) nListCount=nMaxLength-1;
    //人工添加'\0'，确保输出 100%标准 C 字符串
    *(szBuf+nListCount)='\0';
SafePrintf_END_PROCESS:
    return nListCount;
}

```

以上是笔者最常用的一个变参字符串输出函数，可以说笔者写程序中，所有需要变参输出的时候，都是把这段代码拷贝过去，简单修改后使用的。

这里有个重点请大家关注一下，就是函数的返回值，根据 C 语言的定义，所有的 printf 相关的函数，返回的都是实际输出的字符串大小，不包括最后的'\0'，这也是我们设计这类函数所必须遵守的规范。这实际上是一种很方便的设计，注意函数中的这句话：

```

nListCount+=Linux_Win_vsnprintf(szBuf+nListCount,
    nMaxLength-nListCount,szFormat,pArgList);

```

正是因为这种设计，我们才得以可以使用简单的加减定位后续输出的起点，以及不断调整数据输出的边界。这在以后的程序设计中，使用非常方便。

这中间大家可能关注到，使用了一个笔者自定义的函数：Linux_Win_vsnprintf，这实际上是因为 Linux 和 Windows 下，在此处调用的函数并不太一样，为了简化逻辑，笔者特地使用一个宏来实现调用，该宏的原型如下：

```

#ifdef WIN32
    #define Linux_Win_vsnprintf _vsnprintf
#else // not WIN32
    #define Linux_Win_vsnprintf vsnprintf
#endif

```

大家可以看到，其实这个函数非常简单，但它功能却非常完善，在前面章节中，我们曾经专门针对 C 语言的字符串函数做了比较和分析，这个函数完全体现了我们的分析结论，即严格的防御性设计，构造 100%安全的 C 语言字符串，不会出现非法的读写越界，并且，确保最终结果不会超长，以标准'\0'结束等原则。

因此，这段函数最大的好处就是，可以帮助我们放心地实现各种输出，即使某些时候，我们因为笔误或疏忽，导致最终输出的字符串长度超出我们给定的缓冲区，但是由于 nMaxLength 的作用，最终的输出结果会被自动截短，程序不会在此处崩溃。

很多年以前，笔者做游戏程序员时，曾经学到一句名言：“给我一个画点函数，我可以绘制整个世界，给我一个高速画点函数，我可以让整个世界的动起来”。其实，Debug 也一样，“给我一个基本输出函数，我可以输出这个世界上任何信息！”。很多事情其实就是这么简单。

5.2 文本输出

文本输出是 Debug 函数的重要功能，很多时候也是商用工程开发的必要环节。很多 C 程序员入门的第一步，学习的 Hello World，就是典型的文本输出实例。

不过，由于近年来 Windows 等 GUI 开发环境的普及，很多程序员可能不再看重文本输出，这导致程序开发中，缺少了一项重要的 Debug 工具。请注意，以文本方式输出格式化字符串，是 C 和 C++ 程序员的基本功，任何时候都是最重要的。

5.2.1 获得时间戳

首先，我们前面提到，每行 Debug 信息的输出，都必须带有时间戳，这是数据分析的基本要求，这里，我们给出一个常用的时间戳获取函数：

```
//向指定的缓冲区输出一个时间戳字符串
//szBuf: 用户指定的输出缓冲区
//nMaxLength: 用户指定的输出缓冲区尺寸
//返回输出的字符总数 (strlen 的长度, 不包括最后的'\0')
static int GetATimeStamp(char* szBuf,int nMaxLength)
{
    time_t t;
    struct tm *pTM=NULL;
    int nLength=0;
    time(&t);
    pTM = localtime(&t);
    nLength=SafePrintf(szBuf,nMaxLength,"%s",asctime(pTM));
    //这里是个重要的技巧, asctime(pTM) 产生的字符串最后会自动带上回车符,
    //这给后面很多的格式化输出带来不便
    //此处退格结束字符串, 目的是过滤掉这个回车符。
    szBuf[nLength-1]='\0';
}
```

请大家注意一个细节，从这个函数开始，我们已经开始使用前面的 SafePrintf 函数了，这里体现了我们一个很重要的思想，每个模块写出来就是帮助自己工作的，一个模块，一旦成熟，添加到自己的库中，下面立即就开始使用，帮助自己一点点完善自己的工作。最终形成自己一套高可用性的库代码。

反过来理解，进入我们库的每一行代码都必须自己感到有用并且好用，如果没用，则表示它没有存在的必要，应该予以删除。

5.2.2 同时输出到文件和屏幕

由于数据传输模块大多数时候，工作在后台，因此，能有屏幕输出的机会并不多。另外，实际运营时，由于每时每刻都有大量的信令到来或发送，即使能输出到屏幕，屏幕也会快速卷屏，人的肉眼根本看不清楚。

因此，作为 Debug 输出的基本函数，至少应该支持同时输出到文件和屏幕，让操作人员可以后期根据文件内容，实现问题追溯。这里提供基本模块定义：

```
//向指定的缓冲区输出一个时间戳字符串
// szFileName: 用户指定的输出文件
// szMode: 常见的文件打开方式描述字符串，一般建议"a+"
//返回输出的字符总数 (strlen 的长度，不包括最后的'\0')
int dbg2file(char* szFileName,char* szMode, char *szFormat, ...)
{
    //前半段和 SafePrintf 几乎一模一样
    char szBuf[DEBUG_BUFFER_LENGTH];
    char szTime[256];
    int nListCount=0;
    va_list pArgList;
    va_start (pArgList,szFormat);
    nListCount+=Linux_Win_vsnprintf(szBuf+nListCount,
        DEBUG_BUFFER_LENGTH-nListCount,szFormat,pArgList);
    va_end(pArgList);
    if(nListCount>(DEBUG_BUFFER_LENGTH-1))
nListCount=DEBUG_BUFFER_LENGTH-1;
    *(szBuf+nListCount)='\0';
    //在此开始正式的输出到各个目标设备
    GetATimeStamp(szTime,256);
    FILE* fp;
    fp=fopen(szFileName,szMode);
    if(fp)
    {
        nListCount=fprintf(fp,"[%s] %s", szTime,szBuf);    //文件打印
        CON_PRINTF("[%s] %s", szTime,szBuf);                //屏幕打印
        fclose(fp);
    }
    else nListCount=0;
    return nListCount;
}
```

这段代码的目的是将指定的信息，同时输出到一个指定的文件和屏幕。请大家注意，这里我们在输出到屏幕时，并没有直接套用 C 语言的 `printf` 函数，而是使用了一个开关宏。该宏定义如下：

```
#define CON_DEBUG 1
#ifdef CON_DEBUG
#define CON_PRINTF printf
#else
#define CON_PRINTF /\
/printf
#endif
```

这是一个常用的以宏方式取消或者建立代码段的标志。由于我们的程序中需要大量的输出函数，很多在调试阶段需要输出到屏幕，在运营阶段，又不需要，这里我们使用一个简单的手法来实现开关。请注意这句：

```
#define CON_PRINTF /\
/printf
```

我们的目的，是希望关闭屏幕输出时，所有的 `printf` 自动失效，这最好的方法当然是直接把该行语句注释掉。但我们简单的如下定义却是不行的：

```
#define CON_PRINTF //printf
```

因为这样的话，C 语言编译器会认为“//”后面的字符为注释，不予理会，这样 `CON_PRINTF` 会被定义为空，最终产生编译错误。

根据 C 语言的编译原则，“\”是续行符，我们可以简单地通过“\”来破坏编译器的判断，这样，当开关关闭时，`CON_PRINTF` 会被定义为如下形式：

```
#define CON_PRINTF /\printf
```

而 C 语言编译器会自动忽略掉“\”，最终，得到我们希望的定义：

```
#define CON_PRINTF //printf
```

当然，有时候在 Windows 平台下的 VC 开发时，我们可能希望以 VC 标准的输出宏 `TRACE` 来输出，这里也很简单，仅需要将上述宏定义简单修改即可：

```
#define CON_DEBUG 1
#ifdef CON_DEBUG
#define CON_PRINTF TRACE
#else
#define CON_PRINTF /\
/TRACE
#endif
```

5.2.3 文本输出的原则

现在，我们已经拥有了最基本的 `Debug` 输出函数，可以有时间戳，可以同时输出到屏幕和文件，有开关，并且是跨平台兼容的。

但是，请大家注意，如果仅仅有了工具，而没有使用原则，是不够的。没有原则，我们输出信息可能会随心所欲，缺乏一个统一的信息表述格式，造成信息混乱，可读性不强。那样的话，输出的信息也就没什么用处了。

根据笔者的工作经验，我们通常所说的原则，就是输出的信息能尽可能标准化地表述一些基本的常规信息，即信息发生的地点，时间等等。这里简单介绍一下笔者所在项目团队常用的原则：

所有的输出功能函数内部，均不负责输出“\n”回车符，信息需要回行的地方，请用户自行放入“\n”。这个很重要，如果功能函数在内部为每次输出都自动添加回行，则我们很难把几个输出拼接成一个大的格式化字符串，实现最终的整体效果输出。因此，我们规定，**输出是否回行，由使用者决定**。

虽然 C 语言已经提供了 `__FILE__` 和 `__LINE__` 宏，用以标定信息发生的地点，但我们不建议使用，原因是不直观。我们很难从“`file=xxxx.cpp, line=1394`”这类的字符串来直观判断一个信令目前正在经过哪个模块，并且，运维人员由于手中没有源代码，更加无法判断，这类信息输出，在运营时几乎毫无用处。

另外一方面，我们前面在 C 和 C++ 语言无错化设计的章节中已经提到，命名法则是最重要的开发法则，每个类，每个函数，每个模块，应该以命名直接体现其功能和用途，如果无法命名，则表示其毫无用途，没有必要开发它。

因此，我们假设，我们的每个类，函数，都是拥有有意义的命名的，那么，我们通常情况下，会使用“类名+函数名”的方式，来标定信息发生地点，如下所示：

```
dbg2file ("test.dbg","a+","CFileDBBlock::CreateFile(): File DB %s init fail!
please check disk!\n",m_szFileName);
```

大家可以看到，我们这里的含义就很清楚了，文件数据库模块 CFileDBBlock 的 CreateFile 函数发生错误，文件数据库“xxx”（m_szFileName 表示的内容）初始化失败，请检查磁盘。最后回行。

因此，即使是完全没有看过源代码的运营维护人员，在看到上述信息时，也能较为清晰的报告说，某某模块的某某函数好像有点问题，请检查一下。同时，对于有经验的运维人员，也能通过简单的字面含义，做出一些基础的故障定位判断，如上例，运维人员已经可以通过着手检查磁盘空间是否够用来判断故障原因，并通过清理磁盘空间来排除故障。这里笔者和大家约定一下，在后续的库代码和应用工程实例中，将统一使用上述原则规范格式化输出，使信息尽量简单明了。

5.3 二进制输出的 Debug 函数

在前面的章节中，我们已经论述了如何实现文本信息输出的相关内容，但在工程实践中，仅有这种输出还是远远不够的。

在数据传输中，很多时候，为了控制带宽的占用，降低计算机解析的成本，使用的都是二进制信令，而并非文本信令，另一方面，我们常见的树、队列等常见数据结构，内部也使用二进制格式作为数据描述，诸如此类的情况还有很多。

当然，我们可以为每种信令，每种数据结构，都写一个单独的 PrintfInfo 函数，以人可以阅读的方式来描述其内容，但这样开发工作量过大，而且很多时候也没有必要。程序员往往只需要简单观察一下某个关键字节的值，即可做出数据分析判断。

因此，以标准二进制方式输出数据块的信息，就变成 Debug 的一个重要手段。这里，我们说的标准，主要是指约定俗成的如下格式：

```
0000 - 00 00 00 00 00 00 00 00 00 ..... 
```

如上所示，二进制输出主要分为三段，第一段为地址编码，第二段为每个字节的 16 进制表示，第三段是 ASCII 字符表示，按字节表示能以 ASCII 字符显示的地址单元的值。这里给大家列出笔者常用的基本的二进制输出函数模块：

```

//输出格式
//0000 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 xxxxxxxxxxxxxxxxxxxx
//以 ASCII 方式显示数据内容
static int dbg_bin_ascii(char* pPrintBuffer,char* pBuffer,int nLength)
{
    //内部函数，只有笔者本人的函数调用的函数，一般不写防御性设计，保持简洁。
    int i;
    int nCount=0;
    for(i=0;i<nLength;i++) //请关注 for 的写法，笔者所有的 for 都是这个模式
    {
        //ASCII 字符表中，可显示字符代码>32
        if(32<=(pBuffer+i)) //请关注常量写左边
            nCount+=SafePrintf(pPrintBuffer+nCount,256,
                "%c",*(pBuffer+i));
        Else
            //不可显示字符以“.”代替占位，保持格式整齐，且避免出错
            nCount+=SafePrintf(pPrintBuffer+nCount,256,".");
        //如果指定的内容不是可以显示的 ASCII 字符，显示‘.’
    }
    return nCount;
}
//以十六进制方式显示指定数据区内容。
static int dbg_bin_hex(char* pPrintBuffer,char* pBuffer,int nLength)
{
    int i=0;
    int j=0;
    int nCount=0;
    //一个比较复杂的打印循环，虽然很长，但还是很简单
    for(i=0;i<nLength;i++)
    {
        nCount+=SafePrintf(pPrintBuffer+nCount,256,
            "%02X ",(unsigned char)*(pBuffer+i));
        j++;
        if(4==j)
        {
            j=0;
            nCount+=SafePrintf(pPrintBuffer+nCount,256," ");
        }
    }
    if(16>nLength) //每行打印 16 字节
    {
        for(;i<16;i++)
        {
            nCount+=SafePrintf(pPrintBuffer+nCount,256," ");
            j++;
            if(4==j)
            {
                j=0;
                nCount+=SafePrintf(pPrintBuffer+nCount,256," ");
            }
        }
    }
    return nCount;
}

```

```

//*****这是主入口函数
//以 16 字节为一行，格式化输出二进制数据内容。
void dbg_bin(char* pBuffer,int nLength)
{
    int nAddr=0;
    int nLineCount=0;
    int nBufferCount=nLength;
    int n=0;
    char szLine[256];    //行缓冲
    if(0<nLength)
    {
        while(1)
        {
            n=0;
            n+=SafePrintf(szLine+n,256-n,"%p - ",pBuffer+nAddr);
            nLineCount=16;
            if(nBufferCount<nLineCount) nLineCount=nBufferCount;
            n+=dbg_bin_hex(szLine+n,pBuffer+nAddr,nLineCount);
            n+=dbg_bin_ascii(szLine+n,pBuffer+nAddr,nLineCount);
            CON_PRINTF ("%s\n",szLine);
            nAddr+=16;
            nBufferCount-=16;
            if(0>=nBufferCount) break;
        }
        CON_PRINTF ("\n");
    }
    else CON_PRINTF ("dbg_bin error length=%d\n",nLength);
}

```

有了这个工具，大家可以在今后的函数中，对所关心的二进制数据信息，实现输出和观察，这个示例函数主要是输出到屏幕，但大家根据前面的知识，要想改造成输出到文件，也很简单。后面我们将有专门的示例函数。

5.4 核心 Debug 和日志系统的区别

在刚开始设计传输库的时候，笔者把 Debug 和 Log 系统混为一谈，因为二者实在太类似了。功能类似，接口类似，等等。

但随着开发的深入，笔者渐渐发现这么简单归类并不能满足使用需求。

原因很简单，Debug 关注的是程序本次运行发生的时间，因此，没有日志数据库的需求，只要求把本次运行信息保留即可，下次可以自动删除，其面向的用户是程序员。而 Log 是需要记录过去相当长一段时间的运行日志系统，供故障跟踪分析，其面向的用户主要是运维人员。

简单的说，Debug 相对简单，但强调实时高效，对于信息管理部分的要求比较低，没有什么数据库需求。而 Log 系统则相对复杂得多，要有一定的用户友好度，有一定的数据管理需求，有一定的自动增删功能，等等。

在实现时，Debug 模块有上面的基本功能已经能够工作，而 Log 系统则还有诸如数据文件管理，队列管理等更多的功能，最重要的是，Debug 是核心模块，后续的 MemoryPool

和 Queue 等模块都依赖它，而 Log 模块，则需要依赖 MemoryPool 和 Queue 等模块完成功能，因此，把两个模块分开势在必行。

在本章，由于很多后续的功能模块都还没有介绍，因此，我们此处着重介绍 Debug 模块，而 Log 模块则在后文适当的时候，专门介绍。

5.5 统计模块

前面我已经说了，作为一个 Debug 工具模块，仅仅具有输出功能还是远远不够的，因为我们在 Debug 时，不仅仅需要知道程序有没有错误，更需要知道程序运行的效率如何，程序的稳定性如何，以进一步判断选择的算法有没有问题，是否需要调整等。

而这些需求，都需要使用到大量的统计工作，因此，如何尽可能地抽象出一个统计模块，以简便的方法实现基本的统计功能，也是 Debug 模块必须解决的问题。

那么，我们在实际工程实践中，需要哪些统计工作呢？一般说来，统计，无外乎我们如下几种功能：

- 1、我们需要测定某一个工作的开始时间和结束时间，并测定其 Δt 。
- 2、我们可能需要统计在 Δt 内，某项工作做了多少次动作，或者收发了多少字节等。
- 3、我们可能要求出单位时间内，我们做某项动作或者触发数据的能力。即每秒多少字节或者每秒多少次动作。

以上的需求，我们抽象一下，其实可以简单归纳为累加器， Δ 计算，平均值计算三大类，这样，我们归纳统计模块的需求如下：

- 1、统计模块对用户提供的累加器， Δ 计算，平均值计算三大基本统计功能。
- 2、作为 Debug 模块的组成部分，统计模块也必须具有高可用，高稳定性，尽量使用静态的数据结构，不能产生内存碎片和内存泄露。多线程安全。
- 3、统计模块编程应该尽量简化，本身不占用太多 CPU 和内存等系统资源，尽量减小试验误差。
- 4、为了方便使用起见，统计模块也需要提供一些基本的试验算法，如随机数算法，时间触发器等工具，方便统计试验的开展。

5.5.1 累加器

统计模块的累加器相对要求比较简单，一般都是一个整数值的纯加法计算，我们通常使用后文所述的多线程安全的变量中 CMint 类来实现多线程安全统计。

由于 32bit 以上的 CPU，整数的表示都比较大，因此其表示范围在大多数情况下已经够用。

5.5.2 Δ 计算模块

Δ 计算，简单的说，就是需要提供一个起始触发点，一个结束触发点，然后自动给出两次触发点之间的差值。

由于是差值计算，为了有足够的计算精度，我们以无符号长整形，作为 Δ 计算的主要变量类型。这里我们提供一个 Δ 计算的基本源代码：

```

typedef struct _COUNT_SUB_
{
    unsigned long    m_lXBegin;
    unsigned long    m_lXEnd;
}SCountSub;
//获得ΔX
unsigned long SCountSubGetX(SCountSub& CountSub)
{
    return CountSub.m_lXEnd-CountSub.m_lXBegin;
}
//设置初始值
void SCountSubSetBegin(SCountSub& CountSub,unsigned long n)
{
    CountSub.m_lXBegin=n;
}
//设置结束值
unsigned long SCountSubSetEnd(SCountSub& CountSub,unsigned long n)
{
    CountSub.m_lXEnd=n;
    return SCountSubGetX(CountSub);
}

```

如上，我们以结构体 **ScountSub** 来描述统计信息，并设置三个函数，来分别触发开始、结束，以及获得 Δ 。

之所以使用纯 C 编程，主要是考虑到，C++ 的类，编译后效率会略低于 C，在某些统计值很大的时候，这一点点误差可能会被放大，因此，为了满足某些高精度场合的需求，我们最基本的统计函数，使用纯 C 编程。

当然，大多数时候，我们并不需要这么高的精度，为了方便大家调用，这里我们也提供 C++ 的类访问方式。


```

class CCountSub
{
public:
    CCountSub() {m_CountSub.m_lXBegin=0;m_CountSub.m_lXEnd=0;}
    ~CCountSub() {}
public:
    //设置初始值
    unsigned long SetBegin(unsigned long n){return m_CountSub.m_lXBegin=n;}
    //设置结束值
    unsigned long SetEnd(unsigned long n){return m_CountSub.m_lXEnd=n;}
    //获得初始值
    unsigned long GetBegin(void){return m_CountSub.m_lXBegin;}
    //获得结束值
    unsigned long GetEnd(void){return m_CountSub.m_lXEnd;}
    //获得△
    unsigned long GetX(void)
{return m_CountSub.m_lXEnd-m_CountSub.m_lXBegin;}
    //把结束值赋给初始值，相当于△归零
    void E2B(void){m_CountSub.m_lXBegin=m_CountSub.m_lXEnd;}
    //把结束值赋给初始值，然后给结束值赋新值，这在某些循环统计的场合非常有用
    void Push(unsigned long ulNew){E2B();m_CountSub.m_lXEnd=ulNew;}
    //与前文共用的数据描述结构体
    SCountSub m_CountSub;
};

```

请大家关注，我们这里并没有涉及到多线程安全，这是由于在常规的应用中，性能统计通常发生在某一模块内部，同时，为了减小统计误差，基本的统计模块，我们并没有加锁。至于在某些场合需要线程级安全时，我们可以在应用中，再行加锁来解决。

同时，细心的读者可能会观察到，这里的类成员函数实体，全部写在声明处，这是因为根据 C++ 的规范，这样的函数会被当做内联函数处理，相当于 `inline`。在编译时，编译器会把函数的执行代码直接拷贝到调用处，以减少一次函数调用的开销。这样做的目的主要是为了进一步减少统计成本，降低误差，提供尽可能高的统计精度。

另外还有个细节，这是前文所述的典型的工具类写法，即内部所有的成员函数，成员变量全部公开，方便上层程序调用，至于 C++ 的数据屏蔽，私有数据安全性，由上层程序保证。

5.5.3 平均值计算

在工程应用中，我们遇到的最多的平均值计算，通常就是计算每秒发生多少次动作，或者访问多少字节这一类的应用，因此，这里我们给出以时间（秒）为单位的统计模块，大家以后如果有特殊需求，可以根据这个基本模块自行改写。

```

class CDeltaTime
{
public:
    CDeltaTime() { TouchBegin(); }
    ~CDeltaTime() {}
public:
    //Reset 功能，初始化统计时钟，且 Begin=End
    void Reset(void)
    { m_CountSub.SetEnd(m_CountSub.SetBegin(
    (unsigned long)time(NULL))); }
    //触发开始时间计数
    void TouchBegin(void) { m_CountSub.SetBegin((unsigned long)time(NULL)); }
    //触发结束时间技术
    void TouchEnd(void) { m_CountSub.SetEnd((unsigned long)time(NULL)); }
    //利用上文的 Push 功能，实现循环统计。即 Now->End->Begin
    void Touch(void) { m_CountSub.Push((unsigned long)time(NULL)); }
    //获得△
    unsigned long GetDeltaT(void) { return m_CountSub.GetX(); }
    //平均值计算获得每秒钟的操作数，用户以累加器统计总的操作数，传递进来，
    //本函数自动根据内部的△计算平均值，除法计算提供 double 的精度，并规避除 0 错误
    double GetOperationsPreSecond(unsigned long ulOperationCount)
    {
        double dRet=0.0;
        double dCount=(double)ulOperationCount;
        unsigned long ulDeltaSecond=GetDeltaT();
        double dSeconds=(double)ulDeltaSecond;
        if(0==ulDeltaSecond) return dRet;
        return dCount/dSeconds;
    }
    //以前文模块描述的基本统计模块
    CCountSub m_CountSub;
};

```

以上为基本统计类，大家可以关注到，这里同样没有考虑线程锁的保护，不过这个时间平均值统计模块与前文的使用范围不同，根据笔者的工程经验，很多时候，这个时间平均值统计模块，都是贴合应用，在各个应用线程中直接调用。因此，这里笔者再给出加锁后的原型。

```

//带锁的时间平均值统计模块，所有函数功能同上，本层仅提供锁保护
class CDeltaTimeVSLock
{
public:
    CDeltaTimeVSLock() {}
    ~CDeltaTimeVSLock() {}
public:
    void Reset(void)
    { //Reset 为写动作，不能并发
        m_Lock.EnableWrite();
        m_nDeltaT.Reset();
        m_Lock.DisableWrite();
    }
    void TouchBegin(void)
    { //TouchBegin 为写动作，不能并发
        m_Lock.EnableWrite();
        m_nDeltaT.TouchBegin();
        m_Lock.DisableWrite();
    }
    void TouchEnd(void)
    { //TouchEnd 为写动作，不能并发
        m_Lock.EnableWrite();
        m_nDeltaT.TouchEnd();
        m_Lock.DisableWrite();
    }
    unsigned long GetDeltaT(void)
    { //GetDeltaT 为读动作，可以并发读
        unsigned long nRet=0;
        m_Lock.AddRead();
        nRet=m_nDeltaT.GetDeltaT();
        m_Lock.DecRead();
        return nRet;
    }
    double GetOperationsPreSecond(unsigned long ulOperationCount)
    { //GetOterationPerSecond 为读动作，可以并发读
        double dRet=0.0;
        m_Lock.AddRead();
        dRet=m_nDeltaT.GetOperationsPreSecond(ulOperationCount);
        m_Lock.DecRead();
        return dRet;
    }
private:
    CDeltaTime m_nDeltaT; //基本统计因子
    CMultiReadSingleWriteLock m_Lock; //单写多读锁对象
};

```

细心的读者可以观察到，加锁模型非常简单，仅仅是另外建立一个加锁类 **CDeltaTimeVSLock**，内部聚合了 **CDeltaTime** 而已。

这在前面的资源锁章节已经讲过，其好处显而易见，基本的统计模块提供功能，且没有线程安全保护的模块调用，提供高效统计能力，而加锁的类则简单实现一一对应加锁，本身不提供功能，避免一套代码写两遍，降低维护难度。

这里为了进一步提高加锁之后的效率，我们使用了单写多读锁 `CmultiReadSingleWriteLock`，并细分了读写动作，使效率进一步提升。

5.5.4 统计平均值计算

还有一种平均值计算在数据传输中也经常用到，就是统计平均值。

比如我们在一个服务器中，需要统计平均速率，我们定期循环统计传输的字节数，每个统计周期内，我们利用传输的字节数，以及 Δt ，就可以计算出在本周期内的传输速率，而我们将所有统计周期的速率做平均计算，即可获得一个相对比较准确的统计平均值速率。

这个算法，在动态负载均衡算法，和智能择优的算法中很常用，这里我们也给出一个基本的原型代码。读者可以自行发挥，修订。

```
//基本统计平均值模块
typedef struct _COUNT_
{
    SCountSub      m_Sub;    //  $\Delta x$  计算
    unsigned long  m_Sum;    // 统计平均值
} SCount;
//初始化
void SCountReset(SCount& Count,unsigned long n)
{
    Count.m_Sum=n;
    Count.m_Sub.m_lXBegin=0;
    Count.m_Sub.m_lXEnd=0;
}
//计算统计平均值
unsigned long SCountSum(SCount& Count)
{
    unsigned long X=SCountSubGetX(Count.m_Sub);
    if(0==Count.m_Sum) Count.m_Sum=X;        //初值如果为 0,则直接赋值,避免误差
    else Count.m_Sum=(Count.m_Sum+X)/2;      //计算统计平均值
    return Count.m_Sum;
}
//返回 Sum 值
unsigned long SCountGetSum(SCount& Count)
{return Count.m_Sum;}
//返回当前的  $\Delta x$ 
unsigned long SCountGetX(SCount& Count)
{return SCountSubGetX(Count.m_Sub);}
//设置开始
void SCountSetBegin(SCount& Count,unsigned long n)
{SCountSubSetBegin(Count.m_Sub,n);}
//设置结束值,单纯更新,可以多次刷新
unsigned long SCountSetEnd(SCount& Count,unsigned long n)
{return SCountSubSetEnd(Count.m_Sub,n);}
```

如上所示，我们在实际统计工作中，以 `ScountSetBegin` 开始统计，以后可以多次调用 `ScountSetEnd` 来刷新最新的结束时间，并通过 `ScountSum` 获得统计平均值，如果是前面的例子，这时候返回的，已经是平均速率了。

为了简化表述，我们这个原型例子，仅提供无符号长整形的统计平均值表示。

当然，这里我们也给出 C++的调用模型，方便调用。

```
//所有接口与 c 模式一一对应
class CCount
{
public:
    CCount() {SCountReset(0);}
    ~CCount() {}
public:
    void SCountReset(unsigned long n) {::SCountReset(m_Count,n);}
    unsigned long SCountSum(void) {return ::SCountSum(m_Count);}
    unsigned long SCountSetSum(unsigned long n) {m_Count.m_Sum=n;return n;}
    unsigned long SCountGetSum(void) {return ::SCountGetSum(m_Count);}
    unsigned long SCountGetX(void) {return ::SCountGetX(m_Count);}
    void SCountSetBegin(unsigned long n) {::SCountSetBegin(m_Count,n);}
    unsigned long SCountSetEnd(unsigned long n)
{return ::SCountSetEnd(m_Count,n);}
    SCount m_Count;
};
```

5.5.5 辅助功能函数

在常见的性能测试工作中，除了各种统计功能，我们有很多时候，也需要一些辅助功能来协助构建试验环境。这里介绍几个常用的功能函数。

根据 C 语言的常用写法，所有的随机数在使用前，必须初始化随机数种子，即在程序开始的某个角落，执行下面的语句：

srand((unsigned int)time(NULL));

5.5.5.1 获得非零值

很多时候，为了保证我们测试的离散性，我们在获得一个真值的时候，并不简单期望一个 1，这会漏掉很多种可能性，因此，我们常用的一个产生真值的方法，就是强行获得一个非零的随机数。

```
inline int _GetNot0(void)
{
    int nRet=rand();
    if(!nRet) nRet++;          //如果获得的随机数本身为 0，返回 1
    return nRet;
}
```

5.5.5.2 获得 0

如上所示，如果我们期待一个经过某种计算获得的绝对 0 值，也通常使用下面的方法产生。

```
inline int _Get0(void)
{
    int nRet=rand();
    return nRet^nRet;          //以异或方式求的 0
}
```

5.5.5.3 获得给定范围的随机数

其实 C 语言里面，已经针对随机数提供了大量的函数，本来已经可以很方便的调用。不过在工程实践中，笔者发现团队里很多程序员，居然很多时候用错了随机数，造成一些不该有的错误，因此，把随机数的获得函数，也做了标准化封装。

```
//获得给定区间内的随机数
inline int GetRandomBetween(int nBegin,int nEnd)
{
    int n=_GetNot0();          //获得一个随机数
    int nBetween=0;
    if(0>nBegin) nBegin=-nBegin;    //防御性设计防止 Begin 为负值
    if(0>nEnd) nEnd=-nEnd;        //防御性设计防止 End 为负值
    if(nBegin>nEnd)              //调整 Begin 和 End 的顺序, 保证 End>Begin
    {
        nBetween=nEnd;
        nEnd=nBegin;
        nBegin=nBetween;
    }
    else if(nBegin==nEnd)        //如果给定的 Beggin 和 End 相等, 即范围为 0
        nEnd=nBegin+10;        //强行定义范围区间为 10
    nBetween=nEnd-nBegin;        //求的区间
    n=n%nBetween;                //通过求余运算限幅
    n+=nBegin;                   //与 Begin 累加, 确保结果落于 Begin 和 End 之间
    return n;
}
```

5.5.5.4 TimeIsUp

这也是常用的一个功能函数, 就是判定一段时间是否到了。这个算法在超时判断中经常用到。

```
#define TimeSetNow(t) time(&t) //设置给定的 t 为当前时间
//判断是否时间到了
// tLast: 最后一次更新时间
//设定的最大时间
inline bool TimeIsUp(time_t tLast,long lMax)
{
    time_t tNow;
    TimeSetNow(tNow);
    long lDeltaT=(long)tNow-(long)tLast;
    if(lMax<=lDeltaT) return true;
    else return false;
}
```

下面是一个简单的调用示例, 我们先初始化一下最后访问时间, 然后不断循环判断时间是否到了。这个示例函数会等候 10 秒钟后退出。

```
void CheckTimeIsUp(void)
{
    time_t tLast;                //最后一次更新时间
    long lMax=10;                //预设 10 秒钟超时
    TimeSetNow(tLast);           //初始化最后一次访问时间
    while(1)
    {
        if(TimeIsUp(tLast,lMax)) //时间到, 跳出
            break;
        else Sleep(1);
    }
}
```

5.6 CLowDebug 工具类

经过前面的准备工作，我们已经可以开始着手准备第一个工具了，CLowDebug 类，顾名思义，这是低级的 Debug 工具。

为什么这么命名呢，在过去很长一段时间，笔者把 Debug 和 Log 工具是混用的，不过，后来发现一个问题，即 Log 日志功能，随着工程的需要，越来越复杂，越来越智能化，显得很“重”，另外，为了实现强大的功能，它开始对锁、内存池这类底层库开始依赖起来，已经不仅仅是一个简单的输出模块。

这就带来一个问题，笔者的工程库总是在不断维护的，有时候，底层的内存池本身，可能也需要 Debug 功能，而由于 Log 对它的依赖关系，内存池就不能再循环依赖 Log 了，在这种情况下，笔者只有单独拆分一个 LowDebug 模块来实现底层模块的日志输出。

这其实体现了一些很重要的工程库维护原则：

- 1、越简单越通用。
- 2、不够傻瓜的模块通用性往往不好
- 3、不能循环依赖

5.6.1 需求分析

在我们设计 LowDebug 模块时，需要先确定一下需求。由于这个模块完全是为其他模块服务的，因此，笔者本人可以算最终客户，一切需求，以笔者的开发习惯和工程开发需求决定。

经过思考，笔者界定出如下需求：

- 1、LowDebug 工作在系统工程库的最底层，其依赖性必须非常小，最多只能依赖前文的一些基本工具函数和定义，不能依赖以后服务的任何对象。
- 2、LowDebug 必须提供跨线程安全性，保证任何调用情况下，不死机，不崩溃，安全可靠。
- 3、LowDebug 必须非常简单，对调用者非常友好，以适应不同的使用者和使用需求，且简化调用。
- 4、原则上，LowDebug 处理的业务是高频业务，因此，尽量不要向屏幕输出（但要提供该功能），避免屏幕输出太过于混乱，影响正常的业务数据观察。
- 5、LowDebug 必须输出到文件，由于其主要记录每次运行期的中间信息，因此，长期保留无意义，可以考虑只维护一个输出问题，每次运行覆写，避免磁盘开销太大。
- 6、LowDebug 必须提供文本和二进制输出两种手段。
- 7、考虑到以后工程可能运行于多任务操作系统，即可能同时有很多实例运行在一个平台，甚至可能运行在同一个进程空间，因此，本类不能提供全局调用，应该以 C++ 类封装，实现对象调用。避免干涉。

5.6.2 数据边界声明

前文我们已经讲过，任何一个模块的设计，首先考虑其数据边界，没有边界的程序，是危险的程序。因此，我们首先需要考虑如何设定自己的数据边界。

经过思考，笔者认为至少有两个数据边界需要界定，第一是每行能输出的最大文本字符串长度，其次是文件名长度。

```
#define DEBUG_BUFFER_LENGTH 1024 //debug 缓冲区大小
#define TONY_DEBUG_FILENAME_LENGTH 256 //文件名长
```

一般说来，这里有一些经验值，一行文字输出，一般 1024 字符足以，另外，不管是 Windows 还是 Linux，一般文件名+路径名的字符串长度，256 字符比较合适。这可以看做一个工程中常用的定义。

5.6.3 类声明

笔者前文给出的一些工具函数和定义，一般都是全局函数或全局定义形式，这主要考虑上述功能相对独立，彼此干涉不严重，但 LowDebug 不同，很可能在同一个进程中被多次实例化。因此，不能使用任何全局变量区，必须把所有相关变量，以类进行封装，使用时，各自独立实例化成不同对象，方能确保运行安全。

举个例子，比如我们写的一个 ocx 控件，使用了 CLowDebug 模块，但在工程中，如一个 Web 页面中，这个 ocx 空间可能被多次使用，并实例化。

因此，如果 LowDebug 没有写成 C++ 的类形式，而使用 C 的全局变量来管理内部变量，那么，在同一个 IE 下显示的网页中，两个 ocx 实例的内部 LowDebug 模块会无锁保护访问同一全局变量内存区，导致前文所述的线程间访问争用。

这个问题的表现就是本来正确的 ocx，会突然变得不正确，突然出现很多匪夷所思的 bug，并且没有规律性和可重复性，程序员无法查找 bug 的根源。这类 bug 一般很可怕。

因此，作为工程库，只要涉及到变量长期保存使用，一般都定义为类形式，就是为了实现多实例间的内存安全访问，避免系统调用导致的 bug。

这里大家可以看看 CLowDebug 的类声明，还是比较简单的。


```

//回调函数构型
typedef void (*_APP_INFO_OUT_CALLBACK)(char* szInfo,void* pCallParam);
class CTonyLowDebug
{
public:
    //静态工具函数，删除一个文件
    static void DeleteAFile(char* szFileName);
    //过滤掉__FILE__前面的路径名，避免 info 太长
    //从后向前过滤，直到找到一个'\\'或'/'为止，
    //返回该字符下一个字符，就是真实文件名起始位置
    //如果找不到，则返回首字符
    static char* GetTrueFileName(char* szBuffer);
public:
    //主业务函数，输出字符串到文件或控制台，变参设计，方便使用。返回字节数，不计"\0"
    int Debug2File(char *szFormat, ...);
    //二进制输出一段内存区，格式参考前文的 dbg4bin
    void Debug2File4Bin(char* pBuffer,int nLength);
public:
    //构造函数和析构函数
    CTonyLowDebug(char* szPathName,        //路径名
        char* szAppName,                  //文件名
        bool bPrint2TTYFlag=false,        //是否打印到控制台屏幕
        _APP_INFO_OUT_CALLBACK pInfoOutCallback=null, //额外的输出回调
        void* pInfoOutCallbackParam=null); //回调函数参数
    ~CTonyLowDebug();
public:
    //这是一个额外加入的功能，很多时候，应用程序有自己的输出设备，
    //比如 Windows 的一个窗口，这里提供一个回调函数，方便应用层
    //在需要的时候，抓取输出信息，打入自己的输出队列
    //至于 public，是因为可能某些内部模块需要看一下这个指针，同时输出
    _APP_INFO_OUT_CALLBACK m_pInfoOutCallback;
    //所有的回调函数设计者，有义务帮调用者传一根指针
    void* m_pInfoOutCallbackParam;
private:
    bool m_bPrint2TTYFlag;                //内部保留的控制台输出标志
    char m_szFileName[TONY_DEBUG_FILENAME_LENGTH]; //拼接好的路径名+文件名
    CMutexLock m_Lock;                    //线程安全锁
};

```

这里大家可以关注到几点，这个类虽然非常简单，但其考虑非常周详，这也是工程库设计的原则：**功能尽量简单，接口尽量方便，周详。**

1、类声明中，public 和 private 是分段写的，很多程序员，书写的时候，喜欢一个 public 和一个 private 搞定一切，这是个很不好的习惯，类声明视为 api，必须精确，针对相关的几个功能函数，做一个 public 或 private 属性定义，可以有效划分段落，帮助阅读。

2、CTonyLowDebug，笔者的英文名是“Tony.Xiao.”，习惯性在自己的模块前添加自己的英文名。这是因为在商用工程项目组中，很可能别的程序员也有类似的想法，也作出一个 CLowDebug 的类，如果使用通用名，很可能在最后整合时冲突，因此，对于太通用的命名，笔者习惯以英文名，或者姓名的拼音缩写标注，避免潜在的冲突可能。

3、这个类输出了两个静态工具函数，这是笔者的习惯，一段代码只写一次，由于 C++ 的特点，一般说来，一个功能函数如果写在一个类中间，则除非对象实例化本类，否则无法调用。这对调用者来说，显得负担很重，而且，多次实例化同义对象，还有 bug 隐患。因此，对于业务上与本类相关，但与类成员变量没有直接关系的工具函数，笔者习惯以 static 修饰，进行静态输出，以后调用者可以直接使用 CLowDebug:: DeleteAFile(...)调用，使调用显得很轻灵。

4、特别值得一提的是 DeleteAFile 函数，其实每个系统平台和 C 编译器，都提供删除文件的功能，但笔者在此特地做了收拢，即所有需要删除文件的功能均调用这个工具方法，以后如果切换平台需要不同的删除写法，笔者仅需简单修改该函数内容即可。大家可以把这个视为一个函数型宏的调用模型。

5、这个类的构造函数比较复杂，有路径名和文件名，分开输入，这是为了适应各个操作系统平台的特征，由于这个类的特征，每个应用程序一般只实例化一次，因此，这个调用难度并不大，关键业务方法，输出类函数，则谨慎控制调用复杂度，方便调用。

6、这里还有个比较深的考虑，Debug2File 作为变参函数，其参数设计一般都不简单，但由于 C 语言程序员的习惯，对于变参函数熟悉度很高，无形中也显得简单了。这说明，函数构型简单原则，有时候也需要尊重习惯。同时，大家可以发现，笔者从一开始，所有的字符串输出和构造函数，都是一个构型，并且，二进制输出，也是一个构型，这说明简单的另外一个原则：同类函数尽量使用类似接口，形成思维惯性，也可以使程序简化，且不容易出错。

7、回调函数的设计方式，在第五章我们已经介绍过，此处不再赘述，但这个设计本身，体现出底层库的一个特点，很多时候，底层库必须为上层应用提供一些中间切入的机制，以便二次利用数据，否则会给应用程序带来很多不便。而函数回调，则是最简单也最有效的中间切入机制。

5.6.4 类工具函数

笔者习惯把一些每个类都相关，可以抽象出来通用的成员函数，视为类工具函数。如前文所述，包括如下几种：

- 1、构造函数和析构函数
- 2、ICanWork
- 3、IAmWorking
- 4、Start、Stop

5.6.4.1 构造函数

```

CTonyLowDebug::CTonyLowDebug(char* szPathName,
                               char* szAppName,
                               bool bPrint2TTYFlag,
                               _APP_INFO_OUT_CALLBACK pInfoOutCallback,
                               void* pInfoOutCallbackParam)
{
    //保留回调函数指针，供业务函数调用
    m_pInfoOutCallback=pInfoOutCallback;
    m_pInfoOutCallbackParam=pInfoOutCallbackParam;
    //保留输出到屏幕的标志，供业务函数参考
    m_bPrint2TTYFlag=bPrint2TTYFlag;
    //拼接输出文件名
    if(szAppName)
        FULL_NAME(szPathName,szAppName,m_szFileName,"dbg")
    else    //允许不输出到文件
        m_szFileName[0]='\0';
    //先删除上次的运行痕迹，避免两次输出干扰
    DeleteAFile(m_szFileName);
    Debug2File("CTonyLowDebug: Start!\n"); //一个简单的声明，我开始工作了
}

```

这里大家可以看到几个设计要点：一是传输的参数做本对象保存，以方便其他内部成员函数调用，其次是实现了覆写逻辑，将上次运行结果删除。最后还有一点，工程库中每个类启动时，习惯性地通过各种 **Debug** 手段，或者 **Log** 手段，声明一下自己开始工作，这对未来的 **Debug** 很有好处，可以看清楚对象的生死。本类由于本来就是底层的 **Debug** 类，所以调用自己的成员函数输出。

这里笔者用到了一个函数型宏，拼接文件名的 **FULL_NAME**，其定义如下：

//Windows 和 Linux 下有个差别，其路径的间隔符，Linux 下符合 Unix 标准，是“/”，而 Windows 下为“\”，这个小小的宏定义，解决这种纷争。

```
#ifndef WIN32
    #define PATH_CHAR "\\\"
#else // not WIN32
    #define PATH_CHAR "/"
#endif
#define FILENAME_STRING_LENGTH 256 //文件名长度统一为 256 字节
//注意参数的表意，从左至右依次为路径、文件名，构建的全名缓冲区，扩展名
#define FULL_NAME(path,name,fullname,ext_name) \ //注意，宏扩行需要用这个字符
{ \
    if(strlen(path)) \
    { \
        if(strlen(ext_name)) \ //如果有扩展名，按扩展名构建
            SafePrintf(fullname, \
                FILENAME_STRING_LENGTH, \
                "%s%s%s.%s", \
                path, \
                PATH_CHAR, \
                name, \
                ext_name); \
        else \
            SafePrintf(fullname, \ //如果没有扩展名，不输出扩展名
                FILENAME_STRING_LENGTH, \
                "%s%s%s", \
                path, \
                PATH_CHAR,name); \
    } \
    else \
    { \
        if(strlen(ext_name)) \ //如果有扩展名，按扩展名构建
            SafePrintf(fullname, \
                FILENAME_STRING_LENGTH, \
                "%s.%s", \
                name, \
                ext_name); \
        else \ //如果没有扩展名，不输出扩展名
            SafePrintf(fullname, \
                FILENAME_STRING_LENGTH, \
                "%s", \
                name); \
    } \
}
```

严格的讲，这个函数型宏写得有些长了，写成一个全局 inline 函数可能更合适一点，不过，考虑到历史因素，这个宏本来是很短，是一次次的库维护中，不断考虑新的情况，扩充构成的，因此，就成了现在这个样子，各位读者能从这里体会到工程库维护中的尊重原则吗？

5.6.4.2 析构函数

析构函数就非常简单，仅仅根据声明原则，输出一句自己结束的提示。不过，因此我们可以预见，凡是使用了我们工程库的应用程序，一定会产生一个 **dbg** 文件，在这个文件第一句，总是构造函数中打印的 **Start** 提示，而最后一句，一定是本函数的 **Stop** 提示。

```
CTonyLowDebug::~CTonyLowDebug()
{
    Debug2File("CTonyLowDebug: Stop!\n");
}
```

5.6.5 业务函数

本类比较简单，其中仅提供四个业务函数，其中还有两个是静态的。

5.6.5.1 GetTrueFileName

静态函数，其目的就是从一个已经经过 **FULL_NAME** 拼接好的文件全名字符串中，将文件名拆分出来。这个函数本类本身不使用，主要是笔者考虑到，既然我们提供了拼接功能，那么，将来未必没有需要拆分的功能，体现在这里做个逆向功能，方便以后潜在的功能调用。

```
char* CTonyLowDebug::GetTrueFileName(char* szBuffer)
{
    char* pRet=szBuffer;
    int nLen=strlen(szBuffer);
    int i=0;
    for(i=nLen-1;i>=0;i--)        //逆向检索，请注意，这是老代码，不太符合无错化原则
    {
        //基本逻辑，找到右数第一个路径间隔符跳出，以此作为文件名开始点
        if('\\'==*(szBuffer+i)) //考虑到 Windows 的路径间隔符
        {
            pRet=(szBuffer+i+1);
            break;
        }
        if('/')==*(szBuffer+i) //考虑到 Unix 和 Linux 的路径间隔符
        {
            pRet=(szBuffer+i+1);
            break;
        }
    }
    return pRet;        //返回的是找到的点，缓冲区是同一缓冲区，该返回值不需要释放
}
```

本函数功能很简单，这里不多介绍，但希望各位读者关注一个要点，在无错化程序设计方法中，强调不要返回指针，并且尽量遵循“谁申请，谁释放”原则，这个函数是工程库中少有的返回指针的一个函数。

但这个函数返回的指针，是在上层应用程序给出的全名缓冲区中，找到的文件名起始点，是无需释放的，因此，这里使用了指针返回。笔者开发中，一般不会允许下层功能层返回动态申请的指针，因为这样很危险，容易导致 **bug**。

5.6.5.2 DeleteAFile

本函数更为简单，仅仅是一个 **C** 的调用。这个函数的设计，主要是出于调用收口的目的。收口、收拢都是程序员常用的关键词，意思是将某一个调用，统一规定为一个接口

方法或函数，这样当外部环境改变，该调用本身发生改变，可以很方便地一次修改到位，避免到程序的每个调用点去修改，降低 **bug** 率。

```
void CTonyLowDebug::DeleteAFile(char* szFileName)
{
    remove(szFileName);
}
```

5.6.5.3 Debug2File

本函数是本类的主要工作函数，大家可能注意到，这里面也有很多尊重和继承老代码的痕迹，这个模块本身也是经历过多次修改的一个工具。

其中，最突出的就是前文设计的回调函数，是近期刚刚添加的，因此，大家可能能关注到，其使用的内部变量都是临时建立的，目的是尽量减小与以前代码的干涉，避免引发新的 **bug**。这里面也体现了严禁变量转义原则，即新增代码坚决不用过去的变量，最多是只读关系，严禁修改，避免因增加功能而引入 **bug**。

```

int CTonyLowDebug::Debug2File(char *szFormat, ...)
{
    //注意，这个类设计时，还没有内存池概念，因此，动态内存申请，原则上应该避免
    //以免出现内存碎片。此处使用静态数组实现 buffer，在浮动栈建立。
    //这也是为什么这个类必须声明最大输出字符串长度的原因
    char szBuf[DEBUG_BUFFER_LENGTH];           //准备输出 buffer
    char szTemp[DEBUG_BUFFER_LENGTH];           //时间戳置换的中间 buffer
    char szTime[DEBUG_BUFFER_LENGTH];           //时间戳的 buffer
    FILE* fp=null;                             //文件指针，以 c 模式工作
    int nListCount=0;
    va_list pArgList;
    time_t t;
    struct tm *pTM=NULL;
    int nLength=0;
    //这是构建时间戳
    time(&t);
    pTM = localtime(&t);
    nLength=SafePrintf(szTemp,DEBUG_BUFFER_LENGTH,"%s",asctime(pTM));
    szTemp[nLength-1]='\0';
    SafePrintf(szTime,DEBUG_BUFFER_LENGTH,"[%s] ",szTemp);
    //注意，此处开始进入加锁，以此保证跨线程调用安全
    m_Lock.Lock();
    {
        //习惯性写法，以大括号和缩进清晰界定加锁区域，醒目。
        //下面这个段落是从 SafePrintf 拷贝出来的，一个逻辑的可重用性，
        //也可以根据需要，直接拷贝代码段实现，不一定非要写成宏或函数。
        va_start (pArgList,szFormat);
        nListCount+=Linux_Win_vsnprintf(szBuf+nListCount,
            DEBUG_BUFFER_LENGTH-nListCount,szFormat,pArgList);
        va_end(pArgList);
        if(nListCount>(DEBUG_BUFFER_LENGTH-1))
nListCount=DEBUG_BUFFER_LENGTH-1;
        *(szBuf+nListCount]='\0';
        //开始真实的输出
        fp=fopen(m_szFileName,"a+");
        //请注意下面逻辑，由于本函数使用了锁，因此只能有一个退出点
        //这里笔者没有使用 goto，因此，必须使用 if 嵌套，确保不会中间跳出
        if(fp)
        {
            //输出到文件
            nListCount=fprintf(fp,"%s%s",szTime,szBuf);
            if(m_bPrint2TTYFlag)
            {
                //根据需要输出至控制台
                TONY_XIAO_PRINTF("%s%s",szTime,szBuf);
                if(m_pInfoOutCallback)
                {
                    //注意此处，回调输出给需要的上层调用
                    //注意，本段为后加，没有使用前文变量，目前是减少 bug 率
                    char szInfoOut[APP_INFO_OIT_STRING_MAX];
                    SafePrintf(szInfoOut,APP_INFO_OIT_STRING_MAX,
                        "%s%s",szTime,szBuf);
                    m_pInfoOutCallback(szInfoOut, //注意把输出字符串传给回调
                        m_pInfoOutCallbackParam); //透传的指针
                }
            }
            fclose(fp);
        }
        else
        {
            nListCount=0;
        }
    }
    m_Lock.Unlock(); //解锁
    return nListCount; //返回输出的字节数，所有字符串构造和输出函数的习惯
}

```

5.6.5.4 Debug2File4Bin

这是以前文 `dbg_bin` 方式输出二进制数据格式的函数，直接调用了前文的工具函数，这里已经开始重用以前的工作成果。另外，考虑到上层程序一般不需要二进制格式输出，本函数没有实现回调。

```
void CTonyLowDebug::Debug2File4Bin(char* pBuffer,int nLength)
{
    m_Lock.Lock();
    {    //注意加锁区域，以及对前文的代码重用。
        dbg2file4bin(m_szFileName,"a+",pBuffer,nLength);
        if(m_bPrint2TTYFlag) dbg_bin(pBuffer,nLength);
    }
    m_Lock.Unlock();
}
```

5.7 基本 Debug 工具小结

在本章结束之时，笔者已经向大家展示了一些常见的 Debug 思路、手法，性能测试相关的一些工具，并且，为大家的工程库，准备了第一个 `LowDebug` 工具类，为以后相关底层库的开发打下基础。

由于业务的细分，虽然应用层很多 Debug 功能仍然大量使用 Log 功能来实现，但 Log 日志系统已经被从 Debug 工具族中剥离出来，作为独立模块存在，这在后文有专门的介绍。

这里笔者认为有必要提醒大家一句，请在阅读本书时，仔细体会第 5 章所述的无错化程序设计思想，笔者在代码中很多的实现，都是遵循其原则在书写，哪怕一些看似非常累赘的地方，也是在体现尊重前作的原则，笔者希望各位读者在本书中，不仅仅能获得一些有用的代码，而是能够进一步理解代码背后的思想。

第 6 章 锁

- 6.1 二元动作理论
 - 6.1.1 二元动作在 C 语言中的书写特性
 - 6.1.2 面向对象 (OO) 和面向过程 (OP) 的本质差异
 - 6.1.3 二元动作在 C++ 语言中的特殊要求
 - 6.1.4 二元动作开发关注要点
- 6.2 锁对象
- 6.3 多线程安全的变量
 - 6.3.1 CMint 和 CMbool 试验
 - 6.3.2 多线程安全的变量模板
- 6.4 单写多读锁
 - 6.4.1 单写多读锁的来源
 - 6.4.2 单写多读锁 C 语言实现
 - 6.4.2 单写多读锁的 C++ 实现
 - 6.4.3 TonyXiaoMinSleep
 - 6.4.3 单写多度锁安全变量
 - 6.4.4 单写多读锁的真实意义
- 6.5 不可重入锁
 - 6.5.1 需求分析
 - 6.5.2 类实现
 - 6.5.3 使用样例
- 6.6 线程控制锁
 - 6.6.1 线程控制锁的实现
 - 6.6.2 线程控制锁的使用
- 6.7 尽量不用锁

第 6 章 锁

前文已经说过，锁是并行计算的必备工具，而在商用并行计算工程中，锁的使用效率和准确使用，有时候可以说是整个工程成败的关键。因此，建议各位读者仔细阅读本章，体会其中深意。

在过去的工作经历中，笔者曾经亲眼见过，一个工程，由于一个简单的锁错误，而导致工程延期两个月的问题，也曾经见过一个简单的锁死，导致一个程序员连续加班 10 天的结果，锁之所以难以驾驭和掌握，主要的原因有如下几条：

1、锁的使用一定是二元性动作，有加锁，一定要解锁，这在很多人看来天经地义，但很不幸，很多程序员由于没有养成良好的代码习惯，程序中随意跳转、跳出，任意抛出异常，导致在某个逻辑分支上，忘了解锁，引发下次调用的 double lock，导致挂死。

2、同一个类里面，经常对公有函数提供锁保护，但是，很不幸，某些公有函数，出于业务需要，调用了另外一些公有函数，导致 double lock，程序挂死。

3、代码的迁移，拷贝过程中，程序员没有资源锁的概念，对于锁的使用是在调用处随着行为而实施的行为锁，多次复制，拷贝之后，在新的调用处忘了加锁，导致对于某个数据的使用加锁不完整，程序运行结果不可知。

4、程序员没有并行计算的设计思维，脑子里没有使用锁保护安全访问的意识，或者，想当然地认为系统提供的某些资源是安全的，导致关键变量没有加锁，程序运行结果不可知。举个例子，笔者以前就想当然认为 socket 是安全的，没有加锁，结果程序挂死。

5、即使程序员考虑到加锁的需求，也很难去仔细分辨每个锁的工作效率，锁设置的位置，在程序中的作用范围是否合理，最终导致加了锁，但是锁的效率很低，很难满足客户需求的程序效率。

6、锁使用最恐怖的一点，就是一旦出错，体现 bug 的关键点，往往不是错误的根源，二者甚至相隔很远，这样的 bug 几乎不可能查找，很多时候甚至还不如重构来得快。

6.1 二元动作理论

前面我们已经讲过，锁的使用，内存的申请和释放，对象的构造与析构，这些都是二元动作，对于这类动作，最突出的一个特点就是“有始有终”，即执行了第一步动作，一定要执行第二步，否则必然出现 bug。

因此，笔者在此，利用锁的特性，为各位读者讲解一下二元动作的程序开发要点。

6.1.1 二元动作在 C 语言中的书写特性

我们知道，C 语言是典型的面相过程型语言（OP），以函数作为程序组织的基本单元。当我们利用 C 语言开发程序时，每个函数，描述了一段流程，不同的函数，根据调用树形成嵌套关系，共同组成了一个系统所需要的所有应用功能，最终，完成设计需求。

由于 C 语言的函数，基本上就是一个独立的过程，而我们知道，每个程序流程一般都分为三个部分：

1、初始化体，负责准备资源，初始化后续计算所需变量的初始值，以及其他相关准备工作，在无错化程序设计中，通常把防御性设计，参数合理性判断，也归纳为初始化体。

2、执行体，主要实现函数的主要设计功能，并得到最终结果（正确，或者错误）。

3、结束体，主要负责在完成计算工作后，打扫现场，归还释放资源，并向调用者返回合适的计算结果。

可能有的时候，我们出于优化目的，或者程序易读性考虑，会把一个完整的过程逻辑分拆为多个函数来实现，但其总体逻辑万变不离其宗，基本上还是上述三个逻辑模块共同组成。

因此，在 C 语言中，一般说来锁的二元动作的管理比较简单，只要在初始化体进行加锁，结束体解锁，基本就可保证正确无误，不过，根据笔者经验，这里面还是有很多细节需要关注。

1、所有的逻辑模块，只能有一个出口，这在前文无错化程序设计方法中，已经提到，这里不再赘述，这里面涉及到程序模块的组织架构，goto 的使用等知识，请大家仔细体会前文的方法细节。

2、原则上，二元动作，均执行“谁申请，谁释放”原则，对于同一资源，加锁和解锁过程尽量在同一函数内完成，不要拆分成两个函数，避免后续阅读和调试时，由于人为的理解偏差，导致 bug。可以考虑把加锁和解锁动作封装成一个函数，内部调用具体的功能函数，这样，锁壳和业务逻辑分开，可有效防止 bug。如下例：

```
int Func(void)
{
    m_Lock.Lock();
    {
        //加锁区域内部，逻辑太复杂，导致潜在的 bug 可能
        int i=0;
        for(i=0;i<10;i++)
        {
            //...
            if(...)
                goto Func_End;
            Sleep(10);
        }
    }
Func_End:
    m_Lock.Unlock();
    return i;
}
```

这个例子在一个函数内部既处理业务流程，又处理锁逻辑，显得很“重”很累赘，也容易出现 bug。实际工作中，可以考虑分拆成两个函数，一个专门负责业务逻辑，一个专门负责锁处理，这样，既遵守“谁申请，谁释放”原则，又保证了程序的易读性，简化了程序设计，避免潜在的 bug 危害。

```
int FuncDoIt(void) //内部函数专心执行核心业务
{
    int i=0;
    for(i=0;i<10;i++)
    {
        //...
        if(...)
            goto FuncDoIt_End_Process;
        Sleep(10);
    }
FuncDoIt_End_Process:
    return i;
}

void Func(void) //封装函数专心解决安全的锁访问，此处也是调用入口
{
    int nRet=0;
    m_Lock.Lock();
    {
        nRet=FuncDoIt(); //此处以一个简单的函数调用，取代所有复杂逻辑
    } //程序显得非常简练
    m_Lock.Unlock();
    return nRet;
}
```

一般说来，笔者命名时，喜欢以 **DoIt** 为后缀，声明一个专心执行业务逻辑的函数，而函数的原名，则作为入口封装函数名。如本例的 **Func** 是封装函数，**FuncDoIt** 是具体的业务逻辑，这仅仅是一个习惯。各位读者在后续的工程库代码以及实例中，可以看到很多这类命名。

3、对于确实无法分拆的某些特殊情况，由或者，一个逻辑内又包含了申请内存等其他二元动作时，书写时需要特别谨慎对待，一般说来，应该把加锁和解锁，放在过程逻辑的绝对对称处比较稳妥。

这话可能有点费解，简单说就是：“二元动作所标定的作用域，不允许交叉，只能嵌套，这和 C 语言的大括号原则是基本一样的，甚至，比大括号原则更加严厉，不允许跨越多重右括号跳转，即不允许从内层嵌套，跨越中间层嵌套，直接跳到最外层，以避免忘记释放锁的 **bug**”。

例：6.1.1.1

```
void Func(void)
{
    char* pBuffer=malloc(256); //注意这里申请了内存，也是二元动作
    if(!pBuffer) goto Func_Memory_Area_End_Process;
    {
        //这里是内存二元动作的作用域开始
        m_Lock.Lock();
        {
            //加锁作用域开始
            //注意此处，规则比 C 语言的大括号规则更加严厉，
            //不能跨多重大括号跳转，只能逐级跳出
            //避免出现忘记解锁的现象
            if(...) goto Func_Lock_Area_End_Process;
            //请特别关注这个跳转点的选取

            //...
        }
    }
    Func_Lock_Area_End_Process: //加锁作用域结束，锁域内跳出，只能跳至此处
        m_Lock.Unlock();
}
Func_Memory_Area_End_Process: //内存作用域结束
    if(pBuffer)
    {
        free(pBuffer);
        pBuffer=null;
    }
}
```

大家可能注意到，笔者特意用大括号标定了每个二元动作的作用域，这是比较复杂的调用情况，中间特别强调了二元动作作用域不能嵌套，甚至不允许像大括号一样，一次跳出多级大括号，只能逐级跳出。主要就是为了维护二元动作逻辑的完整性，避免 **bug**。

原则上上述写法已经太复杂了，不符合 C/C++ 语言无错化程序设计方法中的简单化原则，在实际的工程中，建议大家遇到类似的复杂情况，请尽量拆分函数，利用上节的封装理论，来显式割裂锁的二元动作维护逻辑与实际业务逻辑，避免出现 **bug**。

4、商用并行工程，很多都有高性能设计要求，因此，在使用锁的原则中，还应该加上尽量减小锁的作用域这一条，这个原因显而易见，锁作用域越小，其内部包含逻辑就越少，当高性能并发时，其他任务因为加锁而等待的时间越少，因此，锁的效率就越高。正是基于这一理念，笔者在本书中专门提出了“资源锁”的概念，即锁与资源绑定，应用程序仅

在资源访问时需要用锁，其余时候，资源均可以自由分配给其他任务使用，这样效率可望最大化。这在后文有更详细的介绍。

6.1.2 面向对象（OO）和面向过程（OP）的本质差异

笔者最开始开发 C 语言应用程序，后来由于工作需要，逐渐转到 C++ 开发，因此，针对面向对象和面向过程的程序设计理念，基本上都是自学的，没有接受什么专业的培训。

在笔者的学习过程中，很多时候，都在思考 OP 和 OO 的本质差异是什么。经过思考，笔者从世界观角度，给二者做出如下定义：

1、OP 是动作优先的程序设计思路，强调程序的主体是动作，这很好理解，客户需求一般都是要求这个程序完成什么功能，做出什么动作，当我们把所有的动作实现，并串起来执行，功能也就完成了。笔者前文所述，程序就是“搬数”，体现的就是这种思想。

2、OO 则是数据优先，我们知道，所谓对象，就是一堆数据极其相关动作的集合，因此，OO 其实是把客户需求抽象化之后的结果，作为 IT 行业的客户需求，虽然要求这样那样的功能，但本质还是要把数据做出某种变换，显示某种表象，改变某种存储方式。因此，OO 的本质，就是所有的动作都围绕数据服务，数据是核心。如前文的“搬数”，在 OO 中就变为“请数据自己搬家”。

为什么会有这么大的差异性呢？这里，我们有必要回顾一下计算机语言的发展历史，我们知道，最开始的计算机是没有什么语言和操作系统的，基本上属于“裸机”。程序员使用汇编语言开发程序。

在汇编语言时代，数据是无所谓“私有”或“公有”的，所有的内存单元，对所有的程序均可见，因此，所有的变量，实际上都是“公有”的。同时，当时计算机内存普遍偏小，各个模块需要内存重用来交替启动，执行功能。

这给大型工程程序设计带来很大困扰，在多个程序员合作时，很难界定出某个内存单元在某个时刻是划归某个程序员的某个模块使用的，这在多人合作的开发模式中，几乎是一场灾难。

有鉴于此，人们开发出了高级语言，第一代高级语言，其中最著名的就是 Pascal、C 等语言（虽然很多时候，人们把 C 归结为低级语言，但在程序开发思路，笔者还是倾向于将之归纳为高级语言，仅仅是提供了某些内存直接操作的灵活性）。这类语言一个显著的特征就是出现了函数等代码模块单位，标定了变量的作用域，最重要的是，提出了“私有变量”这一概念。

由于有了私有变量，每个函数内部的临时变量再也不会和其他函数冲突，各个程序员之间交叉的内存申请不再存在，所有的这些工作，由编译器代为完成，因此，在工程开发中，由于全局变量的混乱使用导致的 bug 大为减少，至少，这使得几万行，几十万行的代码协同开发成为可能。

不过，这随之带来一个问题，就是函数内私有变量，其生命期是随着函数的终结而终止，这就是变量的“短生命周期”问题。在实际工程应用中，我们很多时候，需要一个变量有很长的生命周期，跨越多个函数实现其功能，但是，这个功能又只属于一个程序逻辑，不希望其他逻辑看到。这个需求，利用函数内部私有变量，就无法实现。

虽然 C 语言提出了一个解决方案，即把变量在.c 文件内做成全局变量，但同时修饰成 static 的，即 c 程序模块内部变量，不允许外部模块访问。

但这是程序员行为，不是语言特性，我们知道，程序员是人，是人就有可能犯错误，这个方案治标不治本，没有从根本解决问题。

这个问题一直到第二代高级语言出现，即面向对象的高级语言出现，才得到最终解决。在 C++ 这类 OO 语言中，引入了“对象”的概念，对象的核心是数据，因此，数据天生就

是长期保存的，与函数的生命周期无关，同时，任何对象内数据，均可以声明为 `private`，即私有，语言和编译器保证无法发生未授权访问。彻底实现了“变量短生命周期问题”。

因此，笔者经过学习 OP 和 OO 编程思想，学习 C 和 C++ 这两门语言，得出一个结论：软件编程语言的发展史，其实就是一个“数据私有化”的历史，是数据私有化效果从“有”到“好”的发展历程。

那么，我们再回到我们本章的主题，我们会发现，二元动作特别强调数据私有化，因为私有化的数据，能有效杜绝非法访问，更好地维护二元动作的完整性，能有效避免程序 bug。因此，在二元动作这个领域，C++ 的表现，会比 C 更好。

如前所述，我们知道，C 语言提出的二元动作解决方案，一般都是基于函数内部的流程管控实现的，因此，在处理变量的“长生命周期”问题上并无帮助，下面，我们需要更进一步地理解 C++ 下处理二元动作的方法和原则。

6.1.3 二元动作在 C++ 语言中的特殊要求

在 C++ 语言中，由于有对象，程序员可以显式声明变量的公有和私有属性，同时，对象内部可以嵌入方法函数，“对象内成员变量”和“成员函数内局部变量”被显式区别开，二者分别具有不同的生命周期，因此，数据私有化获得较完美的解决。

尤其是对象的构造函数和析构函数，可以把一个对象相关资源的所有申请和释放动作，简单集中在对象的创建和摧毁两个关键点完成，大大简化了程序设计的难度，降低了程序员开发的强度，同时也避免了很多的 bug。

所有这些对象特征，给二元动作带来很多便利，我们可以简单在构造函数中进行申请所有的动态内存资源，创建子对象等动作，在析构函数中进行释放，这天然就构造了一个“只有一个退出点”的程序模型，使程序员在开发中途，较少考虑二元动作的隐含 bug 问题，开发工作非常轻松。

不过，归结到锁这个问题，笔者发现和其他二元对象还是有一些差异性需要大家关注：

1、锁往往还是和动作相关联的，正是因为多线程访问动作的碰撞，才需要锁对数据的保护。因此，锁的使用更像函数行为，属于“短生命周期”行为。

2、在商用并行工程中，我们出于高性能的需求，要求尽可能缩短锁的作用域，避免太长的锁内处理时间导致的并发效率低下。

3、虽然所有的成员变量是对象内共享的，但是，锁的使用往往不是全对象域的，往往只针对某些特定成员变量。

上述事例表明，虽然 C++ 提供了对象来封装数据，同时方便了二元动作的操作，但是锁的使用，仍然是局部的，不能简单套用 C++ 的特性来解决。比如，我们不太可能做一个对象锁，在一个对象构造时，就将其加锁，直到析构时再解锁，这样固然是安全了，不过程序的效率可想而知，每个对象在其生命期内，有且仅有一个模块能够访问，这个对象变量，和 C 语言的私有变量也没有什么差别了。

因此，即使在 C++ 语言中，我们将锁作为一个成员变量纳入到对象内部，仍然要关注锁使用的“短生命周期”需求，具体到实做，就是前文所述的“资源锁”实例，所有的公有方法启用锁保护，而私有方法不使用，对外提供统一的线程安全接口，对内则尽量减小锁的作用域，以求安全与效率的平衡。

6.1.4 二元动作开发关注要点

综上所述，二元动作，特别是锁的二元动作，在 C 和 C++语言中，是有一定写法的要求的，不能和普通的代码一概而论，否则容易出现 bug。经过笔者思考，主要有以下几个原则需要关注：

6.1.4.1 有头有尾

有头就有尾。一个二元动作，不管在函数内，还是对象内，写了申请，请立即写释放代码，避免以后遗忘。程序书写时，多利用“插入”模式，而不是“修改”模式。

这里举个代码书写的例子，请大家关注，我们以前文 6.1.1 小节的例子为例，即在一个函数内即申请内存，又使用锁，两层二元动作的嵌套逻辑。

第一步，我们先写一个基本的函数，我的习惯是，只要稍微复杂的函数，首先考虑设置 goto 的标签，方便后续的跳转，哪怕最后没有用，将标签再行删除也行，但开始一定要写。

```
void Func(void)
{
    //此处准备插入代码
Func_Memory_Area_End_Process:
    return;
}
```

第二步，写出内存申请的完整逻辑，注意，有始有终，写出申请，立即书写释放，同时，考虑到内存申请失败后的跳转，标签开始起作用。

```
void Func(void)
{
    char* pBuffer=malloc(256); //注意这里申请了内存，也是二元动作
    if(!pBuffer) goto Func_Memory_Area_End_Process;
    {
        //这里是内存二元动作的作用域开始
        //由此开始插入后续代码
    }
Func_Memory_Area_End_Process: //内存作用域结束
    if(pBuffer)
    {
        free(pBuffer);
        pBuffer=null;
    }
    //此处原有的 return 已经无用，在此删除
}
```

第三步，开始书写加锁逻辑，请关注大括号，新增的锁内逻辑跳出标签。

```

void Func(void)
{
    char* pBuffer=malloc(256); //注意这里申请了内存，也是二元动作
    if(!pBuffer) goto Func_Memory_Area_End_Process;
    {
        //这里是内存二元动作的作用域开始
        m_Lock.Lock();
        {
            //加锁作用域开始
            //由此开始插入后续业务逻辑代码
        }
Func_Lock_Area_End_Process:    //加锁作用域结束，锁域内跳出，只能跳至此处
        m_Lock.Unlock();
    }
Func_Memory_Area_End_Process: //内存作用域结束
    if(pBuffer)
    {
        free(pBuffer);
        pBuffer=null;
    }
}

```

如本例所示，当我们每一个函数，都是遵循这一原则书写出来，那么，由于书写的笔误，或者由于程序员搞忘导致的二元逻辑错误，基本都能避免。

6.1.4.2 使用大括号显式标定作用域

笔者的习惯，是利用大括号和缩进，显式标定二元动作的作用域，以保证易读性，这个在上一节的例子中大家可以看到，此处不再赘述。

6.1.4.3 单一出口原则

这个也在 6.1.4.1 的例子中体现得很明确，笔者在无错化程序设计方法中，特别提出对 goto 的使用原则，主要就是为了遵循这个单一出口原则，避免二元动作不完整，导致的 bug。

另外，针对 Java 等纯结构化设计语言程序员，没有 goto 语句可以使用，又或者，某些程序员为了遵守 C++ 的开发规范，确实不太愿意使用 goto 语句，那么，为了遵循单一出口原则，笔者的建议是将程序尽量细分，每个函数仅完成单一简单逻辑，避免过多的分支跳转，来实现单一出口原则。如 6.1.4.1 的例子，我们可以改写成如下形式，以多个函数体来遵循这一原则。


```

//这是原有完成核心业务逻辑的代码，注意 pBuffer 是外部调用者传入
void FuncDoIt(char* pBuffer)
{
    //...
    if(...) return;          //没有办法，由于 goto 的禁用，只能在此 return。
                                //但由于本函数没有二元动作，因此这么写无错误
    //...
    return;
}
//这是执行加锁业务的封装逻辑
void FuncDoItWithLock(char* pBuffer)
{
    m_Lock.Lock();
    {
        FuncDoIt(pBuffer); //注意透传的 pBuffer
    }
    m_Lock.Unlock();
}
//这是主调用入口，体现在 C++，这个函数是公有函数，上面两个为私有
void Func(void)
{
    char* pBuffer=malloc(256); //内存申请
    if(pBuffer)
    {
        //此处体现 pBuffer 的作用域
        FuncDoItWithLock(pBuffer);
        free(pBuffer);
        pBuffer=null;
    }
}

```

6.1.4.4 尽量遵循“谁申请，谁释放”原则

大家在前文的例子中可以发现，不管是 6.1.4.1 的 goto 模式，还是 6.1.4.3 的多函数实现模式，其实都有一个核心思想，就是“**谁申请，谁释放**”。这也是笔者在工程项目中长期强调的一个重要原则。

根据笔者经验，在商用工程中，唯一不能遵守“谁申请，谁释放”原则的特例，就是前文描述的线程启动，远堆传参的情况。其余所有的需求，均可以按照这一原则完成设计。

因此，请各位读者关注，**如果程序无法做到这一原则，通常应该是程序员没有仔细思考，而不是无法实现。**

当然，在对象中，由于很多资源申请是基于对象生命期的，其资源的申请和释放，分别在构造函数和析构函数中，看似不符合这一原则。不过，如果把我们的着眼点从一个函数放大到一个对象，我们发现，这个原则也是正确的。即该资源又哪个对象申请，仍然由哪个对象释放。

6.1.4.5 申请和释放对称放置原则

这也是一个容易被忽视之处。很多程序员，知道在 6.1.4.1 中提出的原则，即有头有尾，中间插入的写法，但是，在多重嵌套式，往往不注意书写的特性，出现了二元作用域交叉的现象，如下例：

```

void Func(void)
{
    char* pBuffer1=malloc(100);
    char* pBuffer2=malloc(50);
    //...
    free(pBuffer1); //注意，这里的释放顺序颠倒，发生作用域交叉
    free(pBuffer2);
}

```

值得一提的是，这种写法通常情况程序是正确的，因为两个变量 `szBuffer1` 和 `szBuffer2`，业务逻辑上的生命周期相同，一般这么写不会有太大的 **bug**，很多程序员关注不到这里。

但是，如果上述二元动作是一个内存申请与一个锁，而锁保护的又是这个内存区域，则会有很大问题，很容易引发 **bug**。因此，一般要求二元动作嵌套，应该具有绝对的对称性，避免交叉。

在 C++ 的类中，也有类似的问题，大家可以关注下面这个例子。

```

class CClass
{
public:
    CClass()                //构造函数
    {
        m_pBuffer1=malloc(100);    //先申请 m_pBuffer1;
        m_pBuffer2=malloc(50);    //再申请 m_pBuffer2;
    }
    ~CClass()              //析构函数，请千万注意下面的释放顺序
    {                       //释放顺序一定与申请顺序相反
        if(m_pBuffer2)     //一定要先释放 m_pBuffer2
        {
            free(m_pBuffer2);
            m_pBuffer2=null;
        }
        //这样，当某些时候，这里插入访问 m_pBuffer1 的代码时，才安全
        if(m_pBuffer1)     //再释放 m_pBuffer1
        {
            free(m_pBuffer1);
            m_pBuffer1=null;
        }
    }
private:
    char* m_pBuffer1;
    char* m_pBuffer2;
};

```

本书写到这里，笔者不由得冒出一头冷汗，这个原则笔者自己可能都没有遵守好，很多工程库中的代码，可能写到这里，就写随手了。好在笔者的代码已经经过了多个商用工程验证，能正确工作，否则就可能需要大改了。

6.1.4.6 对象的构造和析构函数做什么

一般说来，笔者对于构造函数和析构函数，有个明确的定义，即凡是本类的核心资源，放在构造函数中申请，析构函数中释放。

但是，与业务相关的起停动作，尽量不要放到构造和析构函数中，以预防未来的重入和再次启动问题。

一般说来，锁的使用与动作相关，因此，除了某些必要的锁的初始化和摧毁动作，锁的加锁和解锁动作一般不会出现在构造函数和析构函数中。

不过，有时候，笔者喜欢一种比较特别的写法，如下例：

```
class CClass
{
public:
    CClass() {}
    ~CClass()
    {
        m_Lock.Lock();          //一个特别的写法
        //中间什么也不做
        m_Lock.Unlock();
        //...正常的摧毁代码
    }
    void Func(void)
    {
        m_Lock.Lock();
        //...
        m_Lock.Unlock();
    }
private:
    CMutexLock m_Lock; //这是锁对象，无需初始化和释放
};
```

大家可以关注到，在析构函数中，笔者莫名其妙的增加了一次锁的加锁和解锁调用，中间并没有做任何事情。

这是因为，一般需要加锁的对象，都工作在多任务并行环境中，因此，一个对象的所有公有方法，包括摧毁对象，调用析构函数本身，都可能是不同的线程在调用。因此，有时候笔者发现，在程序退出时，特别是 Windows 系统下开发多线程窗口程序，往往会发生一些错误，导致程序最后不是温和退出，而是以一次崩溃结束。

究其原因，笔者发现虽然自己把每个公有方法，如 `Func`，都加上了锁保护，但是，对于析构函数本身，并没有保护，因此，当一个线程毫无阻碍地摧毁一个对象时，可能导致另外一个正在调用该对象方法的线程崩溃。

上述在析构函数中增加一次锁调用的目的，就是利用锁，使析构过程稍微等待一下，当所有其他线程不再访问时，再行进行摧毁过程。

这是一个小技巧，并不能解决所有问题，不过，自从笔者开始使用这个技巧后，程序退出崩溃的现象，基本就没有了。

6.1.4.7 Start 和 Stop 方法的深入理解

由于笔者明确界定了构造函数和析构函数，仅仅完成数据资源的准备工作，但我们知道，很多时候，作为并行系统中，内置线程的主动对象，一般都需要一个显式的启动和停止接口，方便中间多次起停。

因此，在前文笔者提出，为每个有内置线程的对象，设置显式的 `Start` 和 `Stop` 公用接口，提醒调用者，务必启动后，方能正常执行。

当然，有了 `Start` 和 `Stop` 后，自然对象就分为准备态和运行态两种态，也就必须提供 `ICanWork` 方法来标定准备就绪，以及 `IAmWorking` 来标定正在运行。这四个方法由于是公有方法，原则上都要加锁访问。

不过，在具体实做的时候，由于 Start 有防止重入问题，以及有个运行时先后顺序问题，因此一般采用特别的方法保护，这在后文中有“防止重入锁”的实例来说明。而 ICanWork 和 IamWorking，为了保证执行效率，一般采用只读模型回避对锁的使用，因此，反而这几个公有方法，很少涉及到锁。这在后文有更详细的例子讲解。

这里特别提醒的是，启动的内部线程，并不能因为是内部私有函数，就不用锁，反而，由于公有方法的调用，必然是在其它线程内完成，外部线程和内部线程，在本对象的成员变量处交互，因此，类中的内置线程一定要加锁访问内部成员变量。

6.2 锁对象

我们前文中讲到，以数据传输为主工程，本质上是并行计算体系，不同的网络角色之间，同一台网络计算机内部，数据业务都是并发的。这不可避免带来了进程、线程间的数据争用。因此，锁是最重要的并行计算工具。

在前文第四章的工程库准备初步中，笔者已经向大家展示了基本的跨平台锁的构型，此处我们再复习一下。

```
#ifndef WIN32 //windows 下，纯 C 锁操作的方法
#define MUTEX CRITICAL_SECTION //利用临界区实现的锁变量
#define MUTEXINIT(m) InitializeCriticalSection(m)
#define MUTEXLOCK(m) EnterCriticalSection(m)
#define MUTEXUNLOCK(m) LeaveCriticalSection(m)
#define MUTEXDESTROY(m) DeleteCriticalSection(m)
/* non-windows, e.g Linux */
#else // not WIN32 //非 Win 情况下，实现锁操作的方法
#define MUTEX pthread_mutex_t //使用 Linux/Unix 下线程锁实现
#define MUTEXINIT(m) pthread_mutex_init(m, NULL) //TODO: check error
#define MUTEXLOCK(m) pthread_mutex_lock(m)
#define MUTEXUNLOCK(m) pthread_mutex_unlock(m)
#define MUTEXDESTROY(m) pthread_mutex_destroy(m)
#endif
```

同时，笔者为了方便调用，为大家提供了一个 C++ 的构型，自动实现初始化和摧毁。

```
class CMutexLock
{
public:
    CMutexLock(void) {MUTEXINIT(&m_Lock);} //构造函数，初始化锁
    ~CMutexLock(void) {MUTEXDESTROY(&m_Lock);} //析构函数，摧毁锁
public:
    void Lock() {MUTEXLOCK(&m_Lock);} //加锁操作
    void Unlock() {MUTEXUNLOCK(&m_Lock);} //解锁操作
private:
    MUTEX m_Lock; //锁变量（私有）
};
```

6.3 多线程安全的变量

大家看到这里，可能会觉得锁还是比较简单的，不过，就笔者的经验，锁的复杂，并不在其实现，所有系统的锁，无非就是初始化，摧毁，加锁，解锁四个操作，锁真正的难

点，是如何利用上述简单的锁，来应对工程中各种各样加锁的情况。换言之，锁的难点“在用不在写”。

这里面最突出的一个难点，也是用途最广泛的锁使用模型，就是如何保证跨线程变量访问安全。

通常，操作系统给了很多锁的解决方案，比如一个线程，要操作某个线程间共享变量了，就设置一个信号量，然后等待，同时监听另外一个信号量，直到确定该变量没有其余线程在使用，然后安全访问。在这之前，还要先检查这个变量是不是别人正在使用，如果在用，则必须先等待，等等，程序书写非常麻烦。

不过，笔者自从产生了前文所述“资源锁”的思想，发现这个问题其实非常简单，而且，实现跨平台通用也很容易。即思路反过来，把被访问的对象看成是一个变量，利用 C++ 的类对象封装该变量的所有操作，对外提供锁保护的公有访问方法，这样，自然实现了所有的线程安全访问。

6.3.1 CMint 和 CMbool 试验

笔者的“资源锁”思想，其实就是因为这个线程安全访问变量的问题思考而产生的，因此，当产生这个思路后，笔者首先就在多线程安全变量上做实验，验证这个思路。笔者首先选择了我们开发中利用率最高的 int 型和 bool 型做实验。

6.3.1.1 动态内存申请的考虑

在实现多线程安全变量时，笔者首先考虑了其用途，我们知道，很多时候线程需要远堆传参，这涉及到动态内存申请，而如前文所述，出于对内存碎片的担心，我们一般使用内存池来管理动态申请的内存块，而内存池不能提供对 C++ 动态对象的支持。

因此，考虑到多线程安全变量，以后很可能作为一个内存块结构体的内部成员，在线程间传来传去，首先不能单纯提供 C++ 类来完成这个多线程安全变量对象，至少应该先使用纯 C 的方式实现。后期再考虑使用 C++ 来封装。这样，即使是一个变量处于动态内存中，也可以实现多线程安全的保护。

6.3.1.2 C 语言实现的基本功能

如上所述，笔者首先利用 C 语言实现了第一个原型，由于在 C 语言中，本来就没有 bool 这个变量，int 和 bool 都是整形数，因此，笔者同时实现 int 和 bool 变量类型：

```

//这是.h 文件中的声明
typedef struct _MINT_ //这是整型的多线程安全单元
{
    int      m_nValue; //这是整型值
    MUTEX    m_MyLock; //这是实现保护的 C 语言锁变量
}MINT,MBOOL; //int 型和 bool 型同时实现
//初始化一个线程安全变量,同时可以设置值,返回设置值
extern int MvarInit(MINT& (mValue),int nValue=0);
//摧毁一个线程安全变量
extern void MvarDestroy(MINT& (mValue));
//设置一个线程安全变量的值,返回设置的值
extern int MvarSet(MINT& (mValue),int nValue);
//得到一个线程安全变量的值
extern int MvarGet(MINT& (mValue));
//线程安全变量做加法运算,默认+1
extern int MvarADD(MINT& (mValue),int nValue=1);
//线程安全变量做减法运算,默认-1
extern int MvarDEC(MINT& (mValue),int nValue=1);
//这里是.cpp 文件中的实现
int MvarInit(MINT& mValue,int nValue)
{
    MUTEXINIT(&mValue.m_MyLock); //初始化函数主要就是初始化锁变量
    mValue.m_nValue=nValue; //同时赋初值
    return nValue;
}
void MvarDestroy(MINT& mValue)
{
    MUTEXLOCK(&mValue.m_MyLock);
    MUTEXUNLOCK(&mValue.m_MyLock);
    //还记得上面这个技巧吗? 一次看似无意义的加锁和解锁行为
    //使摧毁函数可以等待其他线程中没有完成的访问,最大限度保证安全
    MUTEXDESTROY(&mValue.m_MyLock); //摧毁锁变量
}
int MvarSet(MINT& mValue,int nValue)
{
    MUTEXLOCK(&mValue.m_MyLock);
    mValue.m_nValue=nValue; //锁保护内的赋值
    MUTEXUNLOCK(&mValue.m_MyLock);
    return nValue;
}
int MvarGet(MINT& mValue)
{
    int nValue;
    MUTEXLOCK(&mValue.m_MyLock);
    nValue=mValue.m_nValue; //锁保护内的取值
    MUTEXUNLOCK(&mValue.m_MyLock);
    return nValue;
}
int MvarADD(MINT& mValue,int nValue)
{
    int nRet;
    MUTEXLOCK(&mValue.m_MyLock);
    mValue.m_nValue+=nValue; //锁保护内的累加动作
    nRet=mValue.m_nValue;
    MUTEXUNLOCK(&mValue.m_MyLock);
    return nRet;
}
int MvarDEC(MINT& mValue,int nValue)
{
    int nRet;
    MUTEXLOCK(&mValue.m_MyLock);
    mValue.m_nValue-=nValue; //锁保护内的减法计算
    nRet=mValue.m_nValue;
    MUTEXUNLOCK(&mValue.m_MyLock);
    return nRet;
}

```

```
//为了简化调用，笔者又定义了一批宏缩写
```

```
#define XMI MvarInit  
#define XME MvarDestroy  
#define XMG MvarGet  
#define XMS MvarSet  
#define XMA MvarADD  
#define XMD MvarDEC
```

大家可以看到，笔者为 `int` 这个变量类型提供了 6 种方法，初始化、摧毁、赋值、取值、累加和递减。当然，如果是 `bool` 类型，程序员就不要去调用累加和递减功能了，因为无意义。

由于本原型使用了纯 C 开发，用户需要自行以 MINT 定义一个结构体变量，传入到上述功能函数中，实现功能。

这个结构体变量就很灵活了，既可以是某个函数，某个对象的内部变量，也可以是远堆传参的某个线程参数结构体中的一个成员。

6.3.1.3 C++类实现

当然，如前一小节，如果我们仅仅实现 C 的访问方式，会很不方便，因为每次使用多线程安全的变量，都要初始化一次，用完还要记得摧毁，这是一个典型的二元动作，操作起来非常麻烦。

经过考虑，笔者发现很多时候，我们不需要在动态内存块中使用多线程安全的变量，而仅仅是普通程序中调用，因此，笔者写出了多线程安全变量的 C++ 的表现形式。

```
//这里全部使用内联函数实现
```

```
//请注意，类中所有公有方法均是调用 c 的方法实现。
```

```
class Cmint //int 型多线程安全变量类  
{  
public:  
    Cmint(int nVlaue=0){XMI(m_nValue,nVlaue);}  
    ~Cmint(void){XME(m_nValue);}  
public:  
    int Get(void){return XMG(m_nValue);}  
    int Set(int nValue){return XMS(m_nValue,nValue);}  
    int Add(int nValue=1){return XMA(m_nValue,nValue);}  
    int Dec(int nValue=1){return XMD(m_nValue,nValue);}  
private:  
    MINT m_nValue;  
};  
class CMbool //bool 型多线程安全的变量类  
{  
public:  
    CMbool(bool nVlaue=false){XMI(m_nValue,nVlaue);}  
    ~CMbool(void){XME(m_nValue);}  
public: //只提供 Get 和 Set 方法  
    int Get(void){return XMG(m_nValue);}  
    int Set(bool nValue){return XMS(m_nValue,nValue);}  
private:  
    MBOOL m_nValue;  
};
```

当笔者这么实现完后，马上代入到线程中试验，一切 ok，基于这两个类生成的变量，在跨线程访问中，体现了足够的安全性，这意味着，以这个思路，笔者可以一劳永逸地解决所有跨线程变量争用问题。并且程序书写非常简便。

6.3.2 多线程安全的变量模板

上述实验其实是近年来笔者正在用的工程库代码，在更早一些的时候，2000 年左右，笔者产生“资源锁”思路后，曾经在 VC 下完成过一个模块，即多线程安全的变量模板，在 Windows 的开发时更加好用一些。这里，笔者也将其 share 在这里，方便 Windows 程序员使用。

这个模板，前后书写了差不多 19 遍才完成，并于 2002 年左右发表在中国最大的程序员网站 CSDN 上，截止 2005 年以前，笔者在 Windows 下完成的所有商用工程项目，均是采用该模板完成，笔者的工程，再也没有出现过多线程变量争用的 bug。

近年来，笔者主要开发跨平台应用，由于前文说过，嵌入式系统的内存一般较小，其交叉编译器考虑到内存开销，一般不允许使用过多的 C++ 高级特性，比如模板和多重继承，嵌入式平台实现起来都很困难。因此，近年来这个模板笔者自己已经不再使用。

6.3.2.1 模板定义

```
//这段定义，放在一个叫 xgmvar.h 的头文件中，没有 cpp 文件
#ifndef _XiaoGeMVarTemplateHasBeenDefined_ //为防止多重连接而设定的条件编译项
#define _XiaoGeMVarTemplateHasBeenDefined_
#include <Afxmt.h> //包含 Windows 多线程库的头文件
template <class MVAR_TYPE>
class MVAR
{
private:
    //为了适应所有的变量类型，此处使用动态内存块保存变量区域，
    //因此，内部仅仅保留一个指向变量区域的指针，通过强制指针类型转换实现访问
    char* m_pBegin;
    CCriticalSection m_csLockHandle; //使用 Windows 临界区类实现锁
```



```

public:
    //构造函数使用标准 C++ 的 new 方式申请内存
    MVAR() {m_pBegin=new char[sizeof(MVAR_TYPE)];}
    //析构函数使用标准 C++ 的 delete[] 方式释放内存
    ~MVAR() {delete[] m_pBegin;}
    //获得变量的长度，也就是获得该变量类型的字节宽度（sizeof）
    int GetLength() {return sizeof(MVAR_TYPE);}
    //设置变量的值
    MVAR_TYPE Set(MVAR_TYPE& value)
    {
        m_csLockHandle.Lock(); //所有访问均使用临界区锁保护
        //请注意，所有的拷贝动作，均使用 memcpy，避免变量类型带来的干扰
        memcpy(m_pBegin, (char*)&value, sizeof(MVAR_TYPE));
        m_csLockHandle.Unlock();
        return value;
    }
    //获取变量的值
    MVAR_TYPE Get()
    {
        MVAR_TYPE myValue;
        m_csLockHandle.Lock();
        memcpy((char*)&myValue, m_pBegin, sizeof(MVAR_TYPE));
        m_csLockHandle.Unlock();
        return myValue;
    }
}

```

```

//重载=, 直接赋值符号
MVAR_TYPE operator=(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    myValue=value;
    memcpy(m_pBegin, (char*)&myValue, sizeof(MVAR_TYPE));
    m_csLockHandle.Unlock();
    return myValue;
}
//重载按位否定运算符~
MVAR_TYPE operator~()
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue, m_pBegin, sizeof(MVAR_TYPE));
    myValue=~myValue;
    m_csLockHandle.Unlock();
    return myValue;
}
//重载++运算符(整型)
MVAR_TYPE operator++(int)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue, m_pBegin, sizeof(MVAR_TYPE));
    myValue++;
    memcpy(m_pBegin, (char*)&myValue, sizeof(MVAR_TYPE));
    m_csLockHandle.Unlock();
    return myValue;
}
//重载++运算符, ++运算符需要重载两次
MVAR_TYPE operator++()
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue, m_pBegin, sizeof(MVAR_TYPE));
    myValue++;
    memcpy(m_pBegin, (char*)&myValue, sizeof(MVAR_TYPE));
    m_csLockHandle.Unlock();
    return myValue;
}
//重载--运算符(整型)
MVAR_TYPE operator--(int)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue, m_pBegin, sizeof(MVAR_TYPE));
    myValue--;
    memcpy(m_pBegin, (char*)&myValue, sizeof(MVAR_TYPE));
    m_csLockHandle.Unlock();
    return myValue;
}
//重载--运算符, --运算符也需要重载两次
MVAR_TYPE operator--()
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue, m_pBegin, sizeof(MVAR_TYPE));
    myValue--;
    memcpy(m_pBegin, (char*)&myValue, sizeof(MVAR_TYPE));
    m_csLockHandle.Unlock();
    return myValue;
}

```



```

//重载+运算符，加法计算
MVAR_TYPE operator+(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    myValue+=value;
    m_csLockHandle.Unlock();
    return myValue;
}

//重载-运算符，减法计算
MVAR_TYPE operator-(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    myValue-=value;
    m_csLockHandle.Unlock();
    return myValue;
}

//重载*运算符，乘法运算
MVAR_TYPE operator*(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    myValue*=value;
    m_csLockHandle.Unlock();
    return myValue;
}

//重载/运算符，除法计算
MVAR_TYPE operator/(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    myValue/=value;
    m_csLockHandle.Unlock();
    return myValue;
}

//重载&运算符，and 运算
MVAR_TYPE operator&(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    myValue&=value;
    m_csLockHandle.Unlock();
    return myValue;
}

```

```

//重载|运算符, or 运算
MVAR_TYPE operator|(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue, m_pBegin, sizeof(MVAR_TYPE));
    myValue|=value;
    m_csLockHandle.Unlock();
    return myValue;
}

//重载^运算符, 异或运算
MVAR_TYPE operator^(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue, m_pBegin, sizeof(MVAR_TYPE));
    myValue^=value;
    m_csLockHandle.Unlock();
    return myValue;
}

//重载+=运算符
MVAR_TYPE operator+=(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue, m_pBegin, sizeof(MVAR_TYPE));
    myValue+=value;
    memcpy(m_pBegin, (char*)&myValue, sizeof(MVAR_TYPE));
    m_csLockHandle.Unlock();
    return myValue;
}

//重载-=运算符
MVAR_TYPE operator-=(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue, m_pBegin, sizeof(MVAR_TYPE));
    myValue-=value;
    memcpy(m_pBegin, (char*)&myValue, sizeof(MVAR_TYPE));
    m_csLockHandle.Unlock();
    return myValue;
}

```

```

//重载*=运算符
MVAR_TYPE operator*=(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    myValue*=value;
    memcpy(m_pBegin,(char*)&myValue,sizeof(MVAR_TYPE));
    m_csLockHandle.Unlock();
    return myValue;
}

//重载/=运算符
MVAR_TYPE operator/=(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    myValue/=value;
    memcpy(m_pBegin,(char*)&myValue,sizeof(MVAR_TYPE));
    m_csLockHandle.Unlock();
    return myValue;
}

//重载&=运算符
MVAR_TYPE operator&=(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    myValue&=value;
    memcpy(m_pBegin,(char*)&myValue,sizeof(MVAR_TYPE));
    m_csLockHandle.Unlock();
    return myValue;
}

//重载|=运算符
MVAR_TYPE operator|=(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    myValue|=value;
    memcpy(m_pBegin,(char*)&myValue,sizeof(MVAR_TYPE));
    m_csLockHandle.Unlock();
    return myValue;
}

//重载^=运算符
MVAR_TYPE operator^=(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    myValue^=value;
    memcpy(m_pBegin,(char*)&myValue,sizeof(MVAR_TYPE));
    m_csLockHandle.Unlock();
    return myValue;
}

```

```

//重载>运算符, 比较运算
BOOL operator>(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    BOOL bRet;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    bRet=(myValue>value);
    m_csLockHandle.Unlock();
    return bRet;
}

//重载<运算符, 比较运算
BOOL operator<(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    BOOL bRet;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    bRet=(myValue<value);
    m_csLockHandle.Unlock();
    return bRet;
}

//重载>=运算符, 比较运算
BOOL operator>=(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    BOOL bRet;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    bRet=(myValue>=value);
    m_csLockHandle.Unlock();
    return bRet;
}

//重载<=运算符, 比较运算
BOOL operator<=(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    BOOL bRet;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue,m_pBegin,sizeof(MVAR_TYPE));
    bRet=(myValue<=value);
    m_csLockHandle.Unlock();
    return bRet;
}

```

```

//重载==运算符, 比较运算
BOOL operator==(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    BOOL bRet;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue, m_pBegin, sizeof(MVAR_TYPE));
    bRet=(myValue==value);
    m_csLockHandle.Unlock();
    return bRet;
}
//重载!=运算符, 比较运算
BOOL operator!=(MVAR_TYPE value)
{
    MVAR_TYPE myValue;
    BOOL bRet;
    m_csLockHandle.Lock();
    memcpy((char*)&myValue, m_pBegin, sizeof(MVAR_TYPE));
    bRet=(myValue!=value);
    m_csLockHandle.Unlock();
    return bRet;
}
};
//以下是以此模板, 动态生成的常见变量类型, 这些变量类型在实际工程中
//基本可以替代C语言基本变量类型, 实现各种计算, 同时, 是多线程安全的
typedef MVAR<char>          MCHAR;          //字符型
typedef MVAR<unsigned char> M UCHAR;        //无符号字符型
typedef MVAR<short>         MSHORT;         //短整型
typedef MVAR<unsigned short> MUSHORT;       //无符号短整型
typedef MVAR<int>           MINT;           //整型
typedef MVAR<unsigned int>  MUINT;          //无符号整型
typedef MVAR<long>          MLONG;          //长整形
typedef MVAR<unsigned long> MULONG;         //无符号长整形
typedef MVAR<float>         MFLOAT;         //浮点数值型
typedef MVAR<double>        MDOUBLE;        //双精度浮点数值型
typedef MVAR<BOOL>          MBOOL;          //布尔型
typedef MVAR<LPVOID>        MLPVOID;        //无符号指针型
typedef MVAR<BYTE>          MBYTE;          //二进制字节型
#endif // _XiaoGeMVarTemplateHasBeenDefined_

```

6.3.2.2 线程控制实例

这里, 为了帮助各位读者理解, 笔者再给出一个多线程类的例子, 大家可以体会一下多线程安全的变量模板的使用方式。另外, 这段示例, 对于在类中如何聚合线程函数, 如何利用多线程安全的变量管控线程, 做了初步的展示, 大家也可以体会一下。


```

#include "xgmvar.h" //包含多线程安全变量模板的头文件
class CMyClass //测试类
{
public:
    CMyClass();
    ~ CMyClass();
protected:
    static UINT MyThread(LPVOID pParam); //线程函数
    //大家可以注意，由于我们需要跨线程进行信息通知
    //这类通知变量，一定要线程安全，否则就可能出 bug
    //使用多线程安全变量，由于变量对象内部已经整合了线程安全保护
    //因此，下列对象的访问一定是安全的，开发变得非常简单
    MBOOL m_bThreadContinue; //线程循环标志
    MUINT m_nThreadCount; //线程计数器
};
//类成员实现
CMyClass::CMyClass()
{
    //这里是初始化线程控制变量
    //设置线程持续标志为真，大家注意到没有，
    //这个 MBOOL 和普通的 bool 类型使用完全一样，非常方便
    m_bThreadContinue=TRUE;
    m_nThreadCount=0; //线程计数器归零
    int i=0;
    //开启 100 个线程
    for(i=0;i<100;i++)
    {
        //一定要在开启线程前累加线程计数器，这是因为线程是异步启动的，
        //很可能本循环执行完毕，一个线程都没有启动
        //这样的话，如果一个函数调用本对象，中间没有等待机制
        //则本循环退出后，很可能立即走到下面的析构函数，
        //而由于线程计数器为 0，析构函数中的等待不起作用，导致 bug
        m_nThreadCount++; //注意前文++运算符的重载
        AfxBeginThread(MyThread,this); //开启线程，以本对象指针为参数
    }
}
CMyClass::~CMyClass()
{
    m_bThreadContinue=FALSE; //通知线程关闭
    while(m_nThreadCount!=0) //等待所有线程退出
    {
        Sleep(1); //等待中，合理释放 CPU 时间片，睡眠 1ms
    }
    //此时退出，线程已经全部安全退出!!!
}
UINT CMyClass::MyThread(LPVOID pParam)
{
    CMyClass* pThis=(CMyClass*)pParam; //得到本对象指针
    //注意，随时监视析构函数的退出标志
    while(pThis->m_bThreadContinue.Get()) //唯一不方便的取值操作
    {
        //循环体
        Sleep(100); //合理释放 CPU 时间片
    }
    pThis->m_nThreadCount--; //线程计数器减 1，
    //当该值为 0，表示所有线程退出
    //注意前文--运算符的重载

    return 0;
}

```

关于线程的控制原理，在后文线程池中，有专门讲解，此处不再赘述。大家在此仅需关注一点即可，由于我们有了多线程安全的变量类型，可以很方便地按照常规数据类型的操作来操作，跨线程的变量共享不再是难事，程序显得简洁，高效，并且，足够安全。

再次声明，这个模板仅适用于使用 VC++ 的 Windows 程序员，笔者近年来由于跨平台开发需求，已经不再使用和维护该模板，有兴趣的读者，可以利用本书中的锁知识，将其改写为跨 Linux 和 Windows 的模板使用。不过，嵌入式系统，请尽量不要用这个模板。

6.4 单写多读锁

单写多读锁是一种比较特殊的应用，一般的教科书很难讲到这么细致的东西。但是，在商用工程中，由于有大量高性能服务器的开发需求，开发人员对于锁的效率非常关注，因此，开发出这类锁来使用。这应该是完全在工程中诞生的一个概念。

6.4.1 单写多读锁的来源

前文我们说过，锁最重要的工作，就是在并行计算系统中，对多线程共享的资源进行一种同步保护，即将异步的、并发的访问请求，通过锁的悬挂机制，强行改为同步的，串行的访问，避免同时修改资源内容，导致的读写 bug。

从实施机制上，锁的逻辑，保证了同一时间段，有且仅有一个线程能成功访问资源，其余线程在进入锁作用域的过程中，会被系统悬挂，直到当前加锁线程执行解锁退出操作，才有可能争取到下一次的操作机会。

正是因为这个机制，引出了一句话，**多线程并行系统的同步，一定是以效率降低为代价的**。因为不管多少条线程访问同一共享资源，只有一个线程能够成功，其他的都处于悬挂暂停状态。

这在商用秉性工程的程序员看来，对锁的使用是又爱又恨，不用吧，程序会崩溃，用吧，稍微不注意就会导致性能低下。二者似乎是一种无解的矛盾。

不过，经过笔者分析，发现其实这个问题还可以进一步细分。我们知道，多线程访问共享资源，最大的风险来自于“写”操作，因为写就有修改，有可能有修改中间态，如果此时有其他线程进行读写访问，二者就可能互相干扰，导致错误的结果。锁主要的目的，就是锁住这种“写”操作。

如果一个共享资源是纯“读操作”，其实是不需要用锁的，因为任何时候，它的内容不会改变，都是安全的。

基于这个理念，笔者开始考虑，如果有一种锁，对于“写”很严厉，任何时候，只能有一个“写”操作，写操作进行时，任何其余的“读”和“写”都被禁止，而“读”操作则相对宽松，多个“读”都可以并发，只要这期间不要参杂“写”动作即可。

如果以这个思路实现一把锁，则可望在安全的前提下，把系统的性能调到最大。这，就是“单写多读锁”的由来。

笔者思考了一下，写出了如下的实现方案：

1、单写多读锁内部应该有个 bool 型的“写”操作标志，该标志为真时，所有的读写都应该禁止。

2、单写多读锁内部还应该有个 int 型的“读”计数器，统计当前有多少个“读”操作在并行。当该计数器为 0 时，表示无任何“读”操作。

3、单写多读锁提供“进入写”和“退出写”的方法

a) 进入“写”时，首先检查是否有其他“写”存在，如果有，悬挂等待。当发现“写”操作标志为“假”，则设置该标志为“真”，表示自己希望“写”，其余写操作会相应被悬挂。

b) 此时，“进入写”操作还需要继续等待所有的“读”操作退出，即“读”计数器归零。以保证所有的“读写”操作均被悬挂或退出，确保安全。

c) 当一切就绪，“进入写”操作解除悬挂状态，此时相当于抢到了“写”锁，允许应用程序对资源实施访问。

d) 当应用程序对资源“退出写”时，设置“写”标志为“假”，其余线程继续抢锁。

4、单写多读锁提供“进入读”和“退出读”方法

a) 当应用程序对资源“进入读”时，首先检查“写”标志是否为“假”，如果是“真”，则悬挂等待。

b) 当检查到“写”标志为“假”时，将“读”计数器+1

c) 返回，让上层应用以“读”方式访问资源，访问完毕后，应用程序调用“退出读”方法，将“读”计数器-1。

6.4.2 单写多读锁 C 语言实现

如前所述，既然是锁，我们就要考虑到以后动态内存保护的需求，以及内存池不能支持 C++ 对象的限制，因此，首先要考虑其 C 语言实现。

6.4.2.1 核心数据结构

首先，我们来定义单写多读锁的核心数据结构体：

```
//定义结构体的习惯：typedef 定义出一种新的，针对本结构体的变量类型，方便后续使用
typedef struct _TONY_XIAO_MULTI_READ_SINGLE_WRITE_LOCK_
{
    //注意变量命名，遵循匈牙利命名法在类中的命名规则
    int m_nReadCount;        //读计数器
    bool m_bWriteFlag;       //写标志
    MUTEX m_Lock;             //作为单写多读锁本身，也应该是多线程安全的，
                                //因此，需要增加一个锁变量
}STonyXiaoMultiReadSingleWriteLock;    //这是新的变量类型，可以定义变量
//习惯性做法，定义结构体后，马上定义其尺寸常量，方便后续申请内存方便
const ULONG STonyXiaoMultiReadSingleWriteLockSize=
    sizeof(STonyXiaoMultiReadSingleWriteLock);
```

这里请注意一点，由于本小节是像大家解释“单写多读锁”的设计方法，但这个单写多读锁，我们可以看出，实际上是一个结构体，而内部还包含一个真正的多线程安全锁，来保护这个单写多读锁的安全。

为了避免混淆，我们这里定一下名字，我们用“单写多读锁”来表示我们要实现的这个具有单写多读功能的模块名字，用“内部锁”来表示单写多读锁数据结构内部的多线程安全锁。

程序设计很多时候都是从数据分析开始，当设计完成，具体开始实做的时候，第一步往往就是定义核心数据结构，这在 C 语言中，往往表现为一个结构体，而在 C++ 中，通常表现为对类中成员变量的设计。当然，由于前面我们所说的“粘合类”特性，在商用

数据传输工程开发中，很多时候，即使是使用 C++ 的面相对象方式，第一步仍然是从数据结构体开始设计。这是通常程序实做开始的第一步。

6.4.2.2 初始化和摧毁函数

构造函数和摧毁函数，主要服务于结构体中的锁，因为在很多操作系统中，锁仅仅是一个变量，如果要真实起到锁的作用，需要利用系统访问，赋予其锁的身份，因此，需要有初始化和摧毁动作。

```
//大家可以看看笔者的函数命名习惯，基本遵循匈牙利命名法，即单词首字母大写
//MRSW 是 Multi Read and Signal Write（多读和单写）的缩写
//MRSWLock 前缀表示单写多读锁
//中间一个“_”分割符，后面是函数的功能描述 Create，创建，Destroy，摧毁，等等
void MRSWLock_Create(STonyXiaoMultiReadSingleWriteLock* pLock)
{
    MUTEXINIT(&(pLock->m_Lock));           //初始化内部锁
    pLock->m_nReadCount=0;                   //初始化读计数器
    pLock->m_bWriteFlag=false;               //初始化写标志
}
void MRSWLock_Destroy(STonyXiaoMultiReadSingleWriteLock* pLock)
{
    MUTEXLOCK(&pLock->m_Lock);               //还记得前文的技巧吗？
    MUTEXUNLOCK(&pLock->m_Lock);             //利用一次空加锁解锁动作，规避风险
    MUTEXDESTROY(&(pLock->m_Lock));         //摧毁内部锁
}
```

这里再说明一点笔者关于程序开发的总结：

从结构化程序设计的角度来看，笔者认为面向过程的 C 语言程序和面向对象的 C++ 程序并无太多不同，很多时候，二者均是围绕一个核心数据在服务。二者的差异性，主要体现在函数设计的语法上。

围绕同一类数据进行服务的 C 语言函数，通常需要将数据结构体从外部传入，这在参数列表中可以看到，如本例的 pLock 参数，而 C++ 则由于已经将数据内置在类对象内部，无需再次传递。

C++ 编译器为类中每一个成员函数（不包括 static 静态成员函数），默认提供一根本类类型的 this 指针参数，这是编译器在编译过程中，自动添加在函数调用中的，因此，任何时候，C++ 的类对象爱那个成员函数，可以使用 this 指针来访问本对象核心数据，而无须像 C 一样，由程序员显式声明，显得方便一点。二者函数上最大的差异，也在这里。

6.4.2.3 状态获知函数

状态获知函数，主要供外部应用程序查询本锁当前状态：是否写状态，以及目前的读动作计数等。

```

//获取写状态
bool MRSWLock_GetWrite(STonyXiaoMultiReadSingleWriteLock* pLock)
{
    bool bRet=false;
    MUTEXLOCK(&(pLock->m_Lock));
    {
        bRet=pLock->m_bWriteFlag;
    }
    MUTEXUNLOCK(&(pLock->m_Lock));
    return bRet;
}

//获取读计数器
int MRSWLock_GetRead(STonyXiaoMultiReadSingleWriteLock* pLock)
{
    int nRet=0;
    MUTEXLOCK(&(pLock->m_Lock));
    {
        nRet=pLock->m_nReadCount;
    }
    MUTEXUNLOCK(&(pLock->m_Lock));
    return nRet;
}

```

6.4.2.4 “进入写”操作

进入写操作，可以说是单写多读锁最难书写的一个函数，其中逻辑非常复杂，笔者也是前前后后，边写边调试，用了差不多两天才调试成功。我们来整理一下思路：

- 1、先检查写标志，如果写标志为真，需要循环睡眠等待，再次检测。
- 2、如果没有人写，则需要设置写标志，并进入下一循环，等待所有的读退出。

这中间最核心的难点就是存在一个联动关系，即当我们检测写操作时，首先，这是一个多线程环境，因此，必须利用内部锁加锁后才能访问内部变量，以保护本单写多读锁自身数据的安全。这就给实现单写多读锁本身的逻辑造成了很大的麻烦。因此，程序编写需要非常慎重：

一个线程执行“进入写”的逻辑，首先需要循环检查“写”标志，以便确定是否可以设置“写”状态。

如果程序检测到已经有人设置了写标志，则自己必须循环睡眠等待，此时，睡眠前还必须解除内部锁，否则的话，本单写多读锁的内部单元实际上仍然是不可访问的，即使原来获取写权限的线程完成工作，想要把写标志设置为“假”，也不可能。双方会形成连锁挂死。因此，一旦检测了写标志为“真”，应该立即解除内部锁，开始睡眠等待。

但是，如果检测到写标志为“假”，表示自己可以获得写权限，则一定不能解锁，因为一解锁，别的线程可能会立即切进来，改变状态，前面检测到的状态，马上又不准了，就会出错。此时，必须立即把写标志置成“真”后，再解锁。

但另一方面，设置写标志为“真”后，我们可以肯定，所有期待“写”的线程会被挂住。下面的任务就是循环等待，不断检测“读”标志是否为0了，即所有的读操作是否退出。

这里又有一个关键点，为了保证所有的读线程，在退出时，能顺利将读计数器-1，另外由于这个单写多读锁的所有读写操作，均是通过了内部锁的保护，因此，我们修改完写标志后，必须立刻解除内部锁，给所有的读线程“让位”，即故意让出时间片缝隙给他们，让他们有能力透过内部锁，修改单写多读锁的内部读计数器。

这个逻辑看起来有点绕口，不过没办法，根据笔者的实际书写经验，必须有这么多细节需要关注。

```

//进入写操作函数
void MRSWLock_EnableWrite(STonyXiaoMultiReadSingleWriteLock* pLock)
{
    while(1)                                //死循环等待
    {
        //第一重循环，检测“写”标志是否为“假”，尝试抢夺“写”权
        //请大家放心，即使有其他线程正在写，操作总会完成
        //因此，这个死循环不会永远死等，一般很快就会因为条件满足而跳出
        MUTEXLOCK(&(pLock->m_Lock));        //内部锁加锁，进入锁域
        {
            //在此域内，其他访问本单写多读锁的线程，会因为内部锁的作用被挂住
            //判断是否可以抢夺“写”权利
            if(!pLock->m_bWriteFlag)
            {
                //如果写标志为“假”，即可以抢夺“写”权利
                //注意，此时一定不要解内部锁，应该立即将写标志置为“真”
                pLock->m_bWriteFlag=true;
                //抢到“写”权后，本轮逻辑完毕，解除内部锁后，可以退出
                //关键：请注意，这里多一句解锁命令，为安全跳出使用
                MUTEXUNLOCK(&(pLock->m_Lock));
                //注意退出方式，使用 goto 精确定位
                goto MRSWLock_EnableWrite_Wait_Read_Clean;
            }    //这是 if 域结束
            //此处，“写”标志为真，表示其他线程已经抢到“写”权，
            //本线程必须悬挂等待。
        }    //这是锁域结束
        //等待睡眠时，应该及时释放内部锁，避免其他线程被连锁挂死
        MUTEXUNLOCK(&(pLock->m_Lock));
        //这是一个特殊的 Sleep，下面会讲到
        TonyXiaoMinSleep();
    }
    //这是第一阶段完成标志，程序运行到此，表示“写”权已经被本程序抢到手
MRSWLock_EnableWrite_Wait_Read_Clean:
    //下面，开始等待其他的“读”操作完毕
    while(MRSWLock_GetRead(pLock)) //请务必关注这个调用
    //这是利用公有的取读状态方法来获取信息
    //前文已经说明，这个函数内部，是“资源锁”模型，
    //即函数内进行了内部锁加锁，是线程安全的，
    //退出函数，内部锁自动解锁
    //因此，本循环 Sleep 期间，本函数没有挂住内部锁，
    //其他线程访问的其他公有方法，可以自由修改读计数器的值。
    //这个规避逻辑异常重要！
    {
        //第二重循环，等待所有的“读”操作退出
        //请放心，一旦“写”标志被置为“真”，新的“读”操作会被悬挂，不会进来
        //而老的读操作，迟早会工作完毕
        //因此，这个死循环，总能等到退出的时候，并不是死循环
        TonyXiaoMinSleep();
    }
}
}

```

大家可以看到，这个单写多读锁的“进入写”操作，针对内部锁的悬挂和释放都是经过了精确的计算，不能多，也不能少。笔者在写这段代码时，前后写了大约十几遍，用时差不多两天才写好，因为中间太多的状态要考虑，确实非常困难。

6.4.2.4 “退出写”操作

退出写操作相对比较简单，只需要在内部锁的保护下，将写标志置回“假”即可。

```
void MRSWLock_DisableWrite(STonyXiaoMultiReadSingleWriteLock* pLock)
{
    MUTEXLOCK(&(pLock->m_Lock));
    {
        pLock->m_bWriteFlag=false;
    }
    MUTEXUNLOCK(&(pLock->m_Lock));
}
```

6.4.2.5 “进入读”操作

“进入读”这个操作，比进入写要简单一些，因为读本身可以并发，主要是循环等待“写”操作解除即可。

但这里面也有一个关注要点，就是如果检测出“写”操作为“假”，不能释放内部锁，需要立即将读计数器+1，使之不为0，因为一旦解除内部锁，别的线程可能在这一瞬间切进来，开始写操作，状态就不准了，会导致 bug。

```
//进入读函数，返回当前的读计数器值
int MRSWLock_AddRead(STonyXiaoMultiReadSingleWriteLock* pLock)
{
    while(1)
    {
        //这里是死循环等待，不过，即使是其他线程在进行写操作，
        //操作总会完成，因此，总有机会碰到写标志为“假”，最终跳出循环
        MUTEXLOCK(&(pLock->m_Lock));    //加内部锁，进入锁域
        {
            if(!pLock->m_bWriteFlag)    //检测写标志是否为“假”
            {
                //如果为“假”，表示可以开始读
                //此时，一定要先累加，再释放内部锁，避免由于空隙，
                //导致别的线程错误切入
                pLock->m_nReadCount++;
                MUTEXUNLOCK(&(pLock->m_Lock));
                //返回读计数器的值，
                //注意：这个值可能不一定是刚刚累加的值
                //由于内部锁已经解除，别的读线程完全可能切进来
                //将这个值增加好几次
                return MRSWLock_GetRead(pLock); //这是本函数唯一跳出点
            }
        }
        //如果写标志为“真”，只能循环等待
        MUTEXUNLOCK(&(pLock->m_Lock));
        //使用特殊睡眠，后文有解释
        TonyXiaoMinSleep();
    }
}
```


6.4.2.5 “退出读”操作

退出读相对也比较简单，只需要在内部锁的保护下，将读计数器-1 即可。

```
//返回计数器变化后的结果
int MRSWLock_DecRead(STonyXiaoMultiReadSingleWriteLock* pLock)
{
    int nRet=0;
    MUTEXLOCK(&(pLock->m_Lock));
    {
        //这是一种习惯性保护，递减计算时，如果最小值是 0，总是加个判断
        if(0<(pLock->m_nReadCount))
            pLock->m_nReadCount--;
        //注意，这里是直接获得读计数器的值，看起来，比进入读要准确一点
        nRet=pLock->m_nReadCount;
    }
    MUTEXUNLOCK(&(pLock->m_Lock));
    return nRet;
}
```

6.4.2.6 “读转写”操作

这算是笔者额外的补充，实际上一次都没有用到过，不过，笔者在构建工程库时，考虑到确实有某种可能，一个访问本来是以“读”方式，获得了单写多读锁的使用权，但是，由于某种条件的成立，它需要对该资源进行一次“写”操作，而我们从前面的逻辑可以看到，单写多读锁的“读”方式是并发的，允许多个读同步进行，因此，在“读”方式进行“写”是严重错误的。

此时，应用程序可能会先退出“读”方式，再进入“写”方式，这样效率显得比较低，因此，笔者专门写了一个“读转写”操作函数，来方便这种调用。

```

void MRSWLock_Read2Write(STonyXiaoMultiReadSingleWriteLock* pLock)
{
    while(1)    //死循环，和进入写中的死循环一个道理
    {
        MUTEXLOCK(&(pLock->m_Lock));
        {
            if(!pLock->m_bWriteFlag)
            {
                //注意这里，一旦检测到可以抢夺写权利
                //先把写标志置为“真”
                pLock->m_bWriteFlag=true;
                //切记，由于是读转写，以前进入读的时候，已经把计数器+1
                //这里一定要-1，否则会导致计数器永远不为0，系统挂死
                if(0<(pLock->m_nReadCount))
                    pLock->m_nReadCount--;
                //如前，所有状态设置完成，解除内部锁，跳到下一步
                MUTEXUNLOCK(&(pLock->m_Lock));
                goto MRSWLock_Read2Write_Wait_Read_Clean;
            }
        }
        //解除内部锁等待，前文已经说明
        MUTEXUNLOCK(&(pLock->m_Lock));
        TonyXiaoMinSleep();
    }
    //此处开始等待所有的读退出，同“进入写”的逻辑
MRSWLock_Read2Write_Wait_Read_Clean:
    while(MRSWLock_GetRead(pLock))
    {
        TonyXiaoMinSleep();
    }
}

```

这里还特别说明一点，笔者设置了“读转写”，为什么不设置“写转读”呢？这个原因很简单，因为写权限本来就高于读，写是与其他所有的读写操作都是互斥的，因此，进入写状态后，读当然是安全的，因此不必多此一举。

提示：读转写函数，函数名中间的‘2’，是程序员一种约定俗成的写法，2在英文中，是“two”，谐音是“to”，这里就是代替to的意思，因此，这个函数名的作用是Read To Write，读到写操作的意思。与此对应的还有“4”，英文中为“four”，取谐音“for”的意思，如Driver4KeyBoard，就是Driver For KeyBoard的意思，中文含义：键盘的驱动程序。这些程序界的命名传统，历史非常久远，各位读者有兴趣，不妨了解一下。

6.4.2.7 核心要点

大家看到这几个函数，可能有点莫名其妙，这里笔者再详细论述一下。

1、所有的等待工作，大家可以看到，均使用 while(1)做死循环，这在单线程程序中是不可想象的，但是，在并行环境中，这是很常见的现象，因为共享资源有多个线程在运行，我们的线程死循环等待的时候，别的线程可能会修改变量的值，最终达成条件，使我们的线程跳出死循环退出。

2、实例代码中，单写多读锁和内部锁，可能会造成很大困扰，这里再说明一下，单写多读锁其实是完成单写多读逻辑的封装，其本质还是一个 C 语言的结构体，是数据，因此，它本身也需要线程安全，故此，在内部集成了我们前面所说的线程安全锁。对其内部所有的访问，均需要透过线程安全锁，也就是内部锁完成。

3、不过，由于单写多读锁的特性，需要实现锁之类的悬挂作用，其内部的死循环和 Sleep 语句，就是起到这个作用，但此时的关键要点，就是除了查询的一瞬间，我们利用内部锁，实现安全的“读”操作，死循环的其他时间，应该及时释放内部锁，避免其他线程无法改变内部变量的值。

4、这种对线程安全锁域的快进快出的开发模型，恰恰就是并行计算最重要的时间片互相规避的开发技巧。

6.4.2 单写多读锁的 C++实现

由于我们已经使用 C 语言实现了单写多读锁的所有功能，因此，其 C++模式的套用就变得非常简单，仅有几句代码即可。这种开发模型，很类似前文所述的“粘合类”模型，大家可以慢慢体会一下。

```
class CMultiReadSingleWriteLock
{
public:
    //构造函数和析构函数，自动调用结构体的初始化和摧毁
    CMultiReadSingleWriteLock() {MRSWLock_Create(&m_Lock);}
    ~CMultiReadSingleWriteLock() {MRSWLock_Destroy(&m_Lock);}
public:
    //相应的公有方法，完全是调用 c 语言的函数
    void EnableWrite(void) {MRSWLock_EnableWrite(&m_Lock);}
    void DisableWrite(void) {MRSWLock_DisableWrite(&m_Lock);}
    void Read2Write(void) {MRSWLock_Read2Write(&m_Lock);}
    void DecRead(void) {MRSWLock_DecRead(&m_Lock);}
    void AddRead(void) {MRSWLock_AddRead(&m_Lock);}
    bool GetWrite(void) {MRSWLock_GetWrite(&m_Lock);}
    int GetRead(void) {MRSWLock_GetRead(&m_Lock);}
private:
    //私有结构体
    STonyXiaoMultiReadSingleWriteLock m_Lock;
};
```

6.4.3 TonyXiaoMinSleep

这中间，细心的读者可能看到一个奇怪的函数，TonyXiaoMinSleep，这有别于通常系统提供的 Sleep 函数，这里需要详细给大家说明。

在多任务操作系统运行时，根据用户和进程的权限，操作系统一般把运行级别分为两大类，一类是内核级别，一类是应用程序级别，这在 Windows 下，内核级叫 RING0 级，而应用程序叫 RING3 级，Linux 下则一般分为内核代码和用户代码。

多任务操作系统的内核，最重要的工作，就是调度各个任务时间片的分配，这涉及到大量的守候进程或守候线程在工作。从前文我们知道，任何处于死循环的守候线程，通常需要在空闲循环中做一定的 Sleep，以及时释放时间片，操作系统的守候线程也不例外。

但我们知道，通常应用级的 Sleep，精度都比较低，一般说来，Windows 下的精度一般为 10ms 左右，而 Linux 下的精度比 1ms 略微少一点。这也是 Windows 操作系统不能称之为实时操作系统的主要原因，由于内核做了太多的事情，导致应用层的睡眠精度太低，应用线程无法及时醒来，做一些变化或速度很快的工作，如高频实时采样等。

在单写多读锁中，我们知道，其实悬挂是利用我们自己的死循环完成的，而多任务环境，只要是循环，就一定要通过 Sleep 来释放时间片，避免过多的 CPU 占用。

这就带来一个问题，如果我们仍然使用系统提供给应用层的 Sleep，则因为精度太低，每次最小睡眠时间太长，而导致锁的效率非常低下。比如，我们以 1ms 作为最小精度睡眠，则每次苏醒的时间必然大于 1ms，这表示，任何一个线程只要使用锁被悬挂，则必然会浪费 1ms 以上的时间，这说明，一个锁，最快也只能满足每秒 1000 次左右的切换，这个效率确实太低了。

因此，笔者专门寻找了一下高精度的睡眠功能，由于这个高精度睡眠在 Windows 下实现较为麻烦，还需要和 DirectX、多媒体等 API 打交道，而我们的服务器主要是工作于 Linux 下，因此，笔者主要针对 Linux，开发了一个高精度睡眠的函数 TonyXiaoMinSleep。其代码如下：

```
//高精度睡眠函数，采用内联模式加速调用
inline void TonyXiaoMinSleep(void)
{
#ifdef WIN32
    Sleep(1); //Windows 下不做改变，沿用应用级 Sleep 函数
#else // not WIN32
    //Linux 下，使用 nanosleep 实现高精度睡眠
    struct timespec slptm;
    slptm.tv_sec = 0;
    slptm.tv_nsec = 1000; //1000 ns = 1 us
    if(nanosleep(&slptm, NULL) == -1) usleep(1);
#endif //WIN32
}
//Linux 下普通的睡眠函数对比
#define Sleep(ms) usleep(ms*1000)
```

这个函数效果很好，笔者的一段测试锁效率的代码，在赛扬 1.4G 的一台老笔记本上做 Linux 实验，当采用普通 usleep 时，每秒任务切换频率大约 5500 次左右，一旦使用本函数，切换频率高达 27500 次/s，基本满足了高速线程池的效率需求。

6.4.3 单写多读锁安全变量

单写多读锁虽然是笔者为了提升效率，而在系统锁资源基础上，进一步封装逻辑而实现的一种高效率锁，但其本质还是锁，因此，在实际使用过程中，大家可以完全按照普通锁的方式来看待它。

它最大的特点，仅仅是针对读和写提供了不同的加锁和解锁方法，以区别对待的方式提升效率。

因此，笔者考虑到前文所述的多线程安全的变量，完全可以使用单写多读锁再实现一遍，以最大限度地保证这类变量的使用效率。这里，笔者也提供了 int 和 bool 两种类型供读者参考。

6.4.3.1 CMRSWint

单写多度锁设计的的整型安全变量:

```
//这里是类声明
class CMRSWint
{
public:
    CMRSWint();
    ~CMRSWint(){} //析构函数不做任何事
public:
    int Get(void);
    int Set(int nValue);
    int Add(int nValue=1);
    int Dec(int nValue=1);
    int GetAndClean2Zero(void);
    int DecUnless0(int nValue=1); //如果不是, -1
private:
    int m_nValue;
    CMultiReadSingleWriteLock m_Lock;
};
```

```

//这里是实施部分，请注意各个函数内的锁调用，与前文不同
CMRSWint::CMRSWint()           //构造函数初始化变量
{
    m_Lock.EnableWrite();
    m_nValue=0;
    m_Lock.DisableWrite();
}
int CMRSWint::Get(void)         //得到变量的值
{
    int nRet=0;
    m_Lock.AddRead();           //请关注这里，这是读方式，即这个动作可以并发
    {
        nRet=m_nValue;
    }
    m_Lock.DecRead();
    return nRet;
}
int CMRSWint::Set(int nValue)   //设置变量的值
{
    int nRet=0;
    m_Lock.EnableWrite();       //注意，这里是写方式，表示是串行的
    {
        m_nValue=nValue;
        nRet=m_nValue;
    }
    m_Lock.DisableWrite();
    return nRet;
}
int CMRSWint::Add(int nValue)   //加法运算
{
    int nRet=0;
    m_Lock.EnableWrite();       //写方式，串行
    {
        m_nValue+=nValue;
        nRet=m_nValue;
    }
    m_Lock.DisableWrite();
    return nRet;
}
int CMRSWint::Dec(int nValue)   //减法运算
{
    int nRet=0;
    m_Lock.EnableWrite();       //写方式，串行
    {
        m_nValue-=nValue;
        nRet=m_nValue;
    }
    m_Lock.DisableWrite();
    return nRet;
}

```

6.4.3.2 CMRSWbool

单写多度锁设计的的 bool 型安全变量：

```
//bool 变量的实现更加简单
class CMRSWbool
{
public:
    CMRSWbool() {}
    ~CMRSWbool() {}
public:
    //得到变量的值，或者设置变量的值，均调用整型对象完成
    bool Get(void){return (bool)m_nValue.Get();}
    bool Set(bool bFlag){return m_nValue.Set((int)bFlag);}
private:
    CMRSWint m_nValue; //内部聚合一个上文定义的整型单写多读锁安全变量
};
```

6.4.4 单写多读锁的真实意义

单写多读锁，其本质在于，细分了共享资源的“读”和“写”行为，对于“读”操作，它允许并发，实现高效率的工作，而对于“写”操作，它又提供了足够的安全性，而操作系统提供的普通线程安全锁，由于不区分“读”和“写”动作，一律采用串行模式访问，因此效率远不如单写多读锁。

因此，单写多读锁，能为程序员提供更高效的操作手段。在商用工程实践中，这些特性能有效帮助程序员在保持线程安全访问的前提下，提供比普通系统锁更高的效率，实现效率与稳定性的平衡。

由于这种锁，完全是产生于工程实践，教科书上提及得很少，因此各位读者可能对其使用方向不太熟悉，这里，笔者就个人经验，简单提供一些应用方向，供大家参考：

6.4.4.1 “池”技术的需求

在很多“池”技术中，比如内存池，线程池，由于这些是我们应用系统底层的支撑机构，对系统性能影响很大，因此在稳定的前提下，对性能要求非常高。其中的锁，往往需要使用单写多读锁来完成，而不能使用普通锁。

举个例子，在内存池中，我们往往采用链表来管理空闲的内存块，当应用程序申请内存块时，程序会先在空闲内存表中查询，如果有，则直接返回，如果没有，则向系统申请。这个链表，由于跨线程服务，必须使用锁来保证安全。

而大家可能会注意，这里面有读写两类需求，如果使用普通锁，则大量的查询“读”操作，由于串行逻辑可能效率很低，由此导致“池”模块性能偏低。使用了单写多读锁之后，读操作允许并发，因此，对于占用业务量达到 80~90% 的查询动作，实现了并发操作，大幅度提升了效率。

本书后文的内存池就是典型的例子，使用单写多读锁后，效率是普通锁的 5~10 倍。

6.4.4.2 “Session”管理的需求

商用数据传输工程的服务器，通常面临一对多的客户服务，很多时候，需要利用 Session（会话）来管理客户的一笔“长延时交易”，这个 Session 管理器，往往需要用到单写多读锁。

再举个例子，比如我们网站上一个用户调查服务，大约 10 个 html 页面的信息量，即需要客户端浏览器，依次从服务器取得 10 个调查页面，完成全部调查。

我们知道，浏览器访问 http 服务器，一般是使用 “One-Shut” 工作模型，即发起一次连接，取得一个页面，关闭该连接这种工作模型，中间的 socket 是短连接的。即每个页面会产生一次 TCP 连接，以及其 socket，但多个页面间的 socket 并无关系。

这就带来一个问题，服务器如何得知某一次页面请求，是某个客户要求第几个页面，这需要管理。笔者通常把这种超过一次 socket 生命周期以上的业务交易，称为 “长延时交易”。一般说来，管理这类交易，服务器均需建立 Session（会话）这个逻辑。

比如一次网页调查，由多个问题页面共同组成，我们就可以将其视为一个 Session，Session 数据结构中记录其用户身份，题目做到第几个页面等信息，那么，当某一个浏览器发来请求，我们则可以方便地在 Session 表中查到其用户信息，发出相应的页面给该浏览器，一步步引导用户完成调查任务。

而这个 Session 如何管理？我们作为程序员应该能想到，一般都是动态的链表在管理。即来一个新的客户请求，我们先根据浏览器发来的用户信息，检索 Session 表中是否有该用户数据，如果有，则取出下一页面返回，执行继续下一页的逻辑，如果没有，则需要先在 Session 表中构建一个新的 Session，然后，从第一页开始，发给用户做调查。

由于公网运营的 http 服务器，往往同时为成千上万的用户在服务，这个 Session 表可能会很大，而链表的便利检索，一般比较耗时间，如果中间再使用普通锁，则效率极其低下，严重影响服务器的吞吐量。

此时我们可以看到，这种存在大量检索，少量修改和新建的需求，恰恰非常适合单写多读锁来完成，既保证稳定，又提供查询的高效率。

另外，请大家关注这句话：“检索 Session 表中是否有该用户数据，如果有，则取出下一页面返回，执行继续下一页的逻辑，如果没有，则需要先在 Session 表中构建一个新的 Session”，这是一次典型的 “读转写” 操作，前文笔者提供的 “读转写” 函数，在此可以进一步提升效率。

当然，上述举例有点理想状态，实际上，apache 等 http 服务器一般是使用 Java、PHP 等脚本在完成上述问题调查，后台可能使用 MySQL 数据库在管理 Session，不是内存操作，无需考虑锁。但请大家思考一下，MySQL 数据库，要是想上述逻辑和效率，是否需要 “单写多读锁”？

6.4.4.3 “队列” 管理需求

商用并行工程，很多时候都需要用到队列，其实前一小节的 Session 表，也可以视为队列的一个变种。

很多时候，出于系统灵活性考虑，以及对内存资源的节约使用原则，我们需要使用动态的链表来构建队列。如上例，使用单写多读锁来管理队列，会很大程度上提升队列的性能。

6.4.4.4 其他需求

除此之外，很多时候，我们在设计商用并行系统时，通常需要设定一些各个线程共同可见的状态变量，这些变量统一的特点是修改次数不多，但查询次数很多，且非常重要，需要线程安全保护。此时，可以考虑使用单写多读锁来实现其保护，如前文所述的单写多读线程安全变量类型，即为此需求而开发。

6.5 不可重入锁

不可重入锁也是笔者在工程实践中发现的需求，教科书上没有出现过。

我们知道，锁的基本特性，就是“悬挂”，不同的线程，通过同一把锁，访问共享资源，那么，同一时刻，只能有一个线程可以成功获得访问权，其余线程会被悬挂，就是加锁的函数不会返回，会在内部死循环，直到抢到锁为止，这期间，调用加锁函数的线程，会阻塞在加锁函数中，不会做其他事情。

但是有时候，我们的业务比较宽容，往往并不一定要抢到锁做事情，如果实在抢不到，就返回个失败即可，业务逻辑不会出错。这个时候，锁的“悬挂特性”，就变得有害了。

还有一种情况，就是一个业务函数，很多个线程在并发调用执行，这些业务，需要一个先决条件，比如启动某个模块，但这种启动，只要启动一次就够了，不要多启，否则有害，但也不能不起，否则业务无法继续。

通常这种逻辑，出现在某个底层类对象的 **Start** 函数，所有需要这个对象服务的线程，首先会调用这个 **Start**，然后请求服务。此时，必须有一个机制，检测是否已经启动，没有，则启动，否则，不要启。而这种检查本身，也需要线程安全。这时候，往往需要“不可重入锁”。

不可重入锁一般用户防御性设计，起到资源对象自我保护的作用。

6.5.1 需求分析

简单说来，不可重入锁，就是提供一个 **Set** 方法，允许把一个 **bool** 值，从“假”设为“真”，但有个前提，如果它原来是“假”，设置就可以成功，返回“真”这个结果，但如果原来是“真”，则表示设置不能成功，返回“假”这个结果。

请注意，这个需求和 **CMBool** 的逻辑不一样，不能用 **CMBool** 简单替代。**CMBool** 的 **Set** 方法，是不检测原值的，完全按照给定值，修改内部变量的值，一定能修改成功，这里面没有检测逻辑，也不能根据实际设置情况，返回值来提醒调用者。

6.5.2 类实现

```
//不可重入锁类声明
class CNonReentrant
{
public:
    CNonReentrant();
    ~CNonReentrant(){} //析构函数不做任何事
public:
    //设置为真的时候
    // 如果没有设置进入标志，设置，并返回真
    // 如果已经设置进入标志，不设置，并返回假
    //设置为假的时候，
    // 总是成功并返回真
    bool Set(bool bRunFlag);
private:
    CMutexLock m_Lock; //锁
    bool m_bAlreadRunFlag; //内部的变量值
};
```

```

//不可重入锁类实现
CNonReentrant::CNonReentrant()
{
    //初始化内部变量
    m_bAlreadRunFlag=false;
}
//核心的 Set 函数
bool CNonReentrant::Set(bool bRunFlag)
{
    //请注意，返回结果是设置动作的成功与否，和内部的 bool 变量，没有任何关系
    bool bRet=false;
    if(bRunFlag)    //需要设置为真的逻辑，比较复杂
    {
        m_Lock.Lock();    //进入锁域
        {
            if(!m_bAlreadRunFlag)
            {
                //如果原值为“假”，表示可以设置
                m_bAlreadRunFlag=true; //设置内部 bool 变量
                bRet=true;             //返回真，表示设置成功
            }
            //否则，不做任何事，即内部 bool 变量不做设置，且返回假
        }
        m_Lock.Unlock();    //退出锁域
    }
    else
    {
        //这是设置为“假”的情况
        m_Lock.Lock();    //进入锁域
        {
            m_bAlreadRunFlag=false;    //无条件设置内部 bool 变量为“假”
            bRet=true;                 //由于这也是设置成功，因此返回真
        }
        m_Lock.Unlock();    //退出锁域
    }
    return bRet;
}

```

大家可能注意，作为一个 bool 变量类型的锁，笔者这里没有提供 Get 方法，因为不可重入锁的实际使用中，一般都是用于动态判定一次设置的成功与否，不存在单独求值的情况，无此需求，因此，笔者并未书写 Get 代码。

6.5.3 使用样例

不可重入锁看似复杂，实际上使用起来非常简单，这里，笔者给出一个常见的实例，在以后的工程代码中，各位读者还可以看到大量类似的代码。

```

//示例类
class CClass
{
public:
    CClass();
    ~CClass();
    //本类提供 Start 方法，假定是启动一个线程。
    //但本类的逻辑，只需要一个线程即可，多启动有害
    //即 Start 只能运行一次，不允许重入，此处需要防御性设计
    void Start(void)
    {
        //这是利用不可重入锁进行防御性设计
        //如果是第一次运行，则 Set(true) 一定能成功，结果返回真
        //后续代码可以继续执行
        //如果已经启动过，再次调用本代码，Set(true) 一定不能成功，结果返回假
        //本函数直接返回，后续代码不会被再次执行。
        if(!m_NonReentrant.Set(true))
            return;
        //...
    }
    //Stop 则可以执行多次
    void Stop(void)
    {
        //...
        //在 Stop 的最后一行，将不可重入锁置回“假”。
        m_NonReentrant.Set(false);
    }
private:
    CNonReentrant m_NonReentrant;           //不可重入锁对象
};

```

6.6 线程控制锁

前文我们在《多线程安全的变量模板》一节，已经向大家展示了线程控制的基本手法，这里我们再复习一下。

```

#include "xgmvar.h" //包含多线程安全变量模板的头文件
class CMyClass //测试类
{
Public:
    CMyClass();
    ~CMyClass();
    static UINT MyThread(LPVOID pParam); //线程函数
    MBOOL m_bThreadContinue; //线程循环标志
    MUINT m_nThreadCount; //线程计数器
};
//类成员实现
CMyClass::CMyClass()
{
    //这里是初始化线程控制变量
    m_bThreadContinue=TRUE; //设置线程持续标志为真
    m_nThreadCount=0; //线程计数器归零
    int i=0;
    for(i=0;i<100;i++) //开启 100 个线程
    {
        m_nThreadCount++; //注意前文++运算符的重载
        AfxBeginThread(MyThread,this); //开启线程，以本对象指针为参数
    }
}
CMyClass::~CMyClass()
{
    m_bThreadContinue=FALSE; //通知线程关闭
    while(m_nThreadCount!=0) //等待所有线程退出
    {Sleep(1);}
    //此时退出，线程已经全部安全退出!!!
}
UINT CMyClass::MyThread(LPVOID pParam)
{
    CMyClass* pThis=(CMyClass*)pParam; //得到本对象指针
    //注意，随时监视析构函数的退出标志
    while(pThis->m_bThreadContinue.Get())
    {
        //...
        Sleep(100); //合理释放 CPU 时间片
    }
    pThis->m_nThreadCount--; //线程计数器减 1，
    return 0;
}

```

我们知道，线程一般严禁从外部 kill，否则极易造成内存泄漏，二元动作不完整等 bug，造成系统不稳定。因此，常见的线程控制，一般使用两个变量：

- 1、bool 变量 bThreadContinue，用以通知线程函数退出循环
- 2、int 变量 nThreadCount，用以检查所有线程是否已经退出完毕

这种逻辑一般贯穿于笔者所有的线程控制之中，大家可以看到，使用起来还是很麻烦的，需要关注的要点有十几处，遗漏一处，即造成 bug，这显然不够方便。

因此，笔者就思考，是否可以将这个逻辑进一步封装，构建一个专门用于线程控制的锁，来解决所有问题，答案是肯定的。

6.6.1 线程控制锁的实现

```
//线程控制锁类声明和实现，完全采用内联模型，保证效率
class CThreadManager
{
private:
    //大家可能注意到，线程控制锁，大多数情况下用于查询，
    //只有在线程起停时，才会改写内部的值，
    //因此，此处用的全部是单写多读变量
    CMRSWbool m_bThreadContinue;    //线程持续的标志
    CMRSWint m_nThreadCount;        //线程计数器
    //很多时候，我们做测试代码，可能需要多个线程并发
    //在 Debug 打印输出时，可能会需要一个独立的线程 ID，
    //线程 ID 需要做唯一性分配，因此，将这个分配器做在线程安全锁中，能被所有线程看到
    //与业务无关，仅用于区别打印信息来自于哪个线程，
    //这里提供一个线程 ID 提供者，程序员可以根据需要，在线程中使用，
    //这算是一个内嵌的 debug 友好功能。
    CMRSWint m_nThreadID;
public:
    CThreadManager(){} //由于多线程安全变量对象内置初始化，此处无需初始化
    ~CThreadManager(){CloseAll();} //退出时自动关闭所有线程
    //启动逻辑，其实也是初始化逻辑
    //在使用线程控制锁前，请一定要先调用本接口函数
    void Open(void)
    {
        CloseAll();    //为防止多重启动，先执行一次 Close
        //初始化线程控制变量
        m_bThreadContinue.Set(true);
        m_nThreadCount.Set(0);
    }
    //关闭所有线程逻辑
    void CloseAll(void)
    {
        //这个逻辑已经出现几次，此处不再赘述
        m_bThreadContinue.Set(false);
        while(m_nThreadCount.Get()){TonyXiaoMinSleep();}
    }
    //启动一个线程前，线程计数器+1 的动作
    int AddAThread(void){return m_nThreadCount.Add();}
    //线程退出前，线程计数器-1 动作
    void DecAThread(void){m_nThreadCount.Dec();}
    //查询线程维持变量的值
    bool ThreadContinue(void){return m_bThreadContinue.Get();}
    //获得线程计数器的值
    int GetThreadCount(void){return m_nThreadCount.Get();}
    //分配一个线程 ID，供 Debug 使用
    int GetID(void) {return m_nThreadID.Add()-1;}
};
```

6.6.2 线程控制锁的使用

此处，笔者利用前面的例子，向大家暂时线程安全锁的使用方法：

```
class CMyClass                                     //测试类
{
    CMyClass();
    ~CMyClass();
    static UINT MyThread(LPVOID pParam);           //线程函数
    CThreadManager m_ThreadManager;               //线程控制锁
};
//类成员实现
CMyClass::CMyClass()
{
    //这里是初始化线程控制变量
    m_ThreadManager.Open();                       //开启线程控制锁
    int i=0;
    for(i=0;i<100;i++)                             //开启个线程
    {
        m_ThreadManager.AddAThread();             //线程计数器+1
        AfxBeginThread(MyThread,this);            //开启线程，以本对象指针为参数
    }
}
CMyClass::~CMyClass()
{
    m_ThreadManager.CloseAll();                   //退出所有线程
    //此时退出，线程已经全部安全退出!!!
}
UINT CMyClass::MyThread(LPVOID pParam)
{
    CMyClass* pThis=(CMyClass*)pParam;           //得到本对象指针
    int nMyID=pThis->m_ThreadManager.GetID();      //示例，得到一个 Debug ID
    printf("Thread %d is Start!\n", nMyID);       //debug 打印示例
    //注意，随时监视析构函数的退出标志
    while(pThis->m_ThreadManager.ThreadContinue())
    {
        //...
        Sleep(100);                               //合理释放 CPU 时间片
    }
    pThis->m_ThreadManager.DecAThread();           //线程计数器减，
    printf("Thread %d is Stop!\n", nMyID);        //debug 打印示例
    return 0;
}
```

大家可以看到，由于我们使用了线程控制锁，因此，程序开发相对简化了很多，不需要处理多个变量，分别初始化之类的工作，

这体现出 C/C++无错化程序设计的思想，随时注意向下聚合，把一些常用功能，抽象为一个对象来简化开发。

另外，读者可能关注到，在线程函数中，由于线程控制锁提供 ThreadID 服务，因此，每个线程可以轻松获得一个 Debug ID 供打印输出，本例使用中，大家可以在屏幕上看到 0~99 号线程的打印日志，可能是乱序的，但数量一定是 100 条线程。

特别请大家关注的是，在后续的工程代码中，读者可能会看到两种线程控制方法，一种是原始的两个变量控制法，另一个是线程控制锁控制，这是因为有些代码比线程控制锁更早出现，出于尊重成熟代码的原则，笔者就没有再修改，而是沿用原设计。这个原则请大家关注，不要为了修改而修改代码，避免引入不必要的 bug。

6.7 尽量不用锁

本章的目的，主要是为了给各位读者打下一点并行计算的基础，针对锁的几种常见形态给出实例介绍，但是，正如我们在前文介绍的那样，**用锁的最高境界，其实是不用锁。**

这是因为，锁是并行计算的同步化工具，是通过阻塞实现安全的资源访问，因此，必然带来性能上的损失。

正确的做法，应该是详细地规划系统架构，仔细甄别数据的所有权，保证尽量少的公有数据暴露，当一个数据，仅仅隶属于一个使用者时，即纯粹的私有数据类型，则无需用锁。

这个技术，很难有什么通理可以借鉴，笔者本人也是在不断探索之中，因此，建议各位读者在以后的工作学习中，尽量完善自己的系统分析思维，对数据的边界、所有权养成天生的敏感度，慎重地创建和使用锁资源，才是使网络服务高效、安全运行的正确之道。

第7章 内存及资源管理

- 7.1 内存管理的基本要求
 - 7.1.1 不泄露
 - 7.1.2 不产生碎片
 - 7.1.3 可以自动报警
- 7.2 内存池的核心逻辑—内存栈
 - 7.2.1 内存管理的数学模型
 - 7.2.2 管理模型的优化
 - 7.2.3 关于链表管理的思考
 - 7.2.4 内存块元素
 - 7.2.5 内存栈
- 7.3 内存指针注册管理模块
 - 7.3.1 内存注册模块原理介绍
 - 7.3.2 模块设计及类声明
 - 7.3.3 构造函数和析构函数
 - 7.3.4 Add 函数
 - 7.3.5 Del 函数
 - 7.3.6 Modeify 函数
 - 7.3.7 PrintInfo 函数
 - 7.3.8 内存注册模块的深入使用
- 7.4 Socket 注册管理模块
 - 7.4.1 类声明
 - 7.4.2 构造函数和析构函数
 - 7.4.3 Add 函数
 - 7.4.4 Del 函数
 - 7.4.5 PrintInfo 函数
- 7.5 内存池类
 - 7.5.1 类声明
 - 7.5.2 构造函数和析构函数
 - 7.5.3 内存栈公有方法
 - 7.5.4 指针管理方法
 - 7.5.5 Socket 管理方法
 - 7.5.6 PrintInfo 方法
- 7.6 内存管理的深层次含义
 - 7.6.1 资源重用的理念
 - 7.6.2 注册和反注册机制
 - 7.6.3 静态资源的管理思路
- 7.7 被动池的常见组织形式
 - 7.7.1 被动池的数据特性及需求分析
 - 7.7.2 动态与静态被动池的差异性
 - 7.7.3 静态被动池实施原理
 - 7.7.4 被动池的常见组织形式

第 7 章 内存及资源管理

C 语言是公认的一门中低级语言，主要的原因就是其提供了类似于汇编语言的指针调用，将编译器和操作系统内部的很多核心机密，向应用程序公开，使我们的程序，可以自由地使用动态内存申请，指针管理等操作系统级的功能，实现强大的程序能力。

这实际上是把操作系统对内存的管理，向程序员做了公开，程序员可以站在系统的角度，进行动态的内存资源调度，这给 C 和 C++ 程序员带来了莫大的方便性，获得了强大的控制能力，但同时，也给 C 和 C++ 程序带来了天生的安全隐患。

因此，作为一个商业化的 C 和 C++ 程序员，首先就需要熟练掌握对内存，以及各种系统资源的操作能力，能做到不泄露、不溢出、安全使用。这对程序员的综合实力，提出了较高的要求。笔者在本章，将为大家展示对系统内存，以及各个系统资源实施管控的综合技巧和原则。

7.1 内存管理的基本要求

其实很多高级语言，如 Java、Python，都有自己的内存管理器，应用程序一般尽管使用变量即可，程序员很少关心变量失效之后的摧毁问题，更无需关心内存的优化使用，减少内存碎片等细节。

不过，C 和 C++ 语言，把系统内存直接暴露给程序员使用，看似提升了灵活性和方便性，但同时，也放弃了更高级的内存管控机制，这对程序员提出了很高的理论和实战能力的要求，稍有不慎，即会出现 bug。

笔者经过多年的分析，认为如果要彻底杜绝内存相关的 bug，实现 C 和 C++ 语言无错化程序设计，程序员有必要在 C 和 C++ 提供的基本内存操作的基础上，自行构建一个更加合理的内存管理机，帮助程序员实现内存的安全、高效访问。

这个内存管理机，笔者称之为“内存池”，本小节即试图论述其基本的设计需求以及解决方案。

7.1.1 不泄露

由于 C 和 C++ 语言中，内存的申请和释放，一定是二元动作，需要程序员显式地调用相关函数，对称地完成内存操作，才能保证不泄露内存。

这对很多情况下的程序开发，提出了较高的要求，笔者前文花了大量的篇幅，向大家介绍二元动作操作的常见手法，以期避免内存泄漏等 bug。

不过，这些动作一般都是程序员的行为，我们知道，程序员是人，是人就有可能犯错误，纯粹的手工操作规范，并不足以杜绝内存 bug 的产生。

于是笔者就设想，如果在 C 和 C++ 传统的内存管理机制之外，我们自行构建一种内存的管理机制，能在程序员忘了释放内存时，主动替其释放，则可望大大减少内存相关的 bug。因此，内存池的第一个设计目标，是主动替程序员完善二元动作，确保“不泄露内存”。

针对这个问题，笔者通常的解决方案是内部建立一套登记机制，记录所有在用的内存块，当程序退出时，如果发现还有内存块在内存池中处于激活状态，即表示有内存块忘了释放，内存池会帮助程序员释放内存，避免产生内存泄漏。

7.1.2 不产生碎片

前文我们已经说过，对于 7*24 小时运行的服务器和嵌入式设备，其对内存的管理要求很高，仅仅不泄露内存是不够的，还必须保证无内存碎片的产生，确保内存池可以长期有效地提供服务。

从前文我们知道，内存碎片的根源，在于一个程序，无序地申请任意大小的内存，最后导致系统堆上的内存不连贯，虽然从统计上得知，还有足够的内存空间，但这是由小块的内存区域组成，没有足够的，整块的大型空间，使后续的大块内存申请无法成功，导致服务无法继续。

这个问题比较难解决，很多解释型高级语言，可以在运行前，主动分析程序，对大型的内存申请实行预分配制度，但是，在 C 和 C++ 里面，由于是编译执行，动态内存申请又过于灵活，很多内存块的尺寸取决与中间计算结果，因此，无法实现预分配，内存的申请和释放动作完全依赖应用程序自己的设计，无法实现统一管控。

笔者经过思考，总结了如下几条推论，以此来试图控制内存碎片的产生：

1、一个应用程序，总的来说，其使用的所有内存，不会超过其目标运行平台的基本内存空间，这很好理解，每个程序员做程序，总是能预估自己的程序，需要多大内存，因此，在产品说明书上，会提示用户准备足够内存的计算机。

2、由上可知，我们在动态内存块可以重用的前提下，可以利用一套机制，屏蔽内存区的动态释放工作，即所有的动态内存，一旦申请，在本次运行期不再释放，这并不会导致计算机内存溢出。

3、动态内存块可以重用，需要保证两点，首先是有一套管理机制，可以记录申请和释放的所有内存块，把释放的内存块二次提供给新的内存申请使用，其次，必须对内存块取模，减小内存块的种类，提高内存块的可重用性。关于取模的原则和方法，我们后文讨论。

这样的话，如果按照上述机制来设计内存池，虽然我们应用程序在运行过程中，存在大量的动态内存申请，但就该程序运行期总的需求来说，使用的最大内存数并没有多大变化，在操作系统看来，这个程序从一开始，申请了差不多的内存之后，就不再申请，全部是内部重用，自然也就没有内存碎片的产生了。

提示：如果计算机系统内存太小，这种机制导致内存溢出了，那是说明应用程序对自己使用的最大内存没有估计准确，用户使用的计算机太低档，正确的解法不是改程序，而是请用户加大内存，或者干脆换更高档的计算机设备。

7.1.3 可以自动报警

在解决“不泄露”的问题时，笔者设计了一个内存管理链表，在设计该链表的时候，笔者突发奇想，由于内存池已经收拢了所有的内存申请和释放行为，那么，我们自然可以很轻易地知道是哪个模块在申请内存，为什么不把这个信息记录下来，帮助 Debug 呢？

我们知道，内存泄漏问题之所以难以解决，并不是这个问题有多复杂，而是由于内存申请和释放，是程序行为，我们只有在最后的程序运行中，隐约观察到内存使用在不断增长，由此推论可能程序有内存泄漏，但这种观察很不直观，无法帮助程序员精确查找是哪个模块发生了内存泄漏。

但如果我们设计内存池时，要求申请内存的模块，必须注明自己的模块名，在退出时，一旦检测出哪个模块忘了释放内存，马上将其申请信息打印出来，不就可以帮助程序员很方便地检查出发生内存泄漏的模块吗？顺便，也就能在开发期快速解决掉所有的内存泄漏，彻底杜绝这类 bug。

经过思考，笔者的内存池所有的内存申请动作，全部改为如下格式：

```
MemPool.Malloc(int nSize, char* szInfo);
```

这明显有别于 C 语言原有的内存申请函数：

```
malloc(int nSize);
```

这里的 `szInfo`，是笔者规定的 124Bytes 的说明性文字，强迫所有申请内存的模块，必须在其中声明自身的身份，一旦发生泄漏，任何一次运行完毕，内存池析构时，立即会打印出相关的信息，程序员即可实现快速查找。

经过试用，这样的效果非常明显，任何一段程序，只要忘了释放内存，则在第一运行结束时，内存池会自动打印报警信息，信息中标明是哪个模块的哪个函数，由于什么原因分配的内存块忘了释放，程序员几乎立即就可以找到故障点，排除 bug。

提示：笔者在前不久带领团队开发的一个服务器集群中，由于引入了这类注册+自报警机制，所有的 C 和 C++ 程序模块，从未出现内存泄漏，极大地提升了程序的稳定性，也为项目的顺利完成打下了坚实的基础。

7.2 内存池的核心逻辑—内存栈

7.2.1 内存管理的数学模型

内存块如果要提升可重用性，必须对内存块尺寸进行取模，否则的话，很容易因为几个 Bytes 的偏差，导致内存块无法重用，被迫向系统频繁申请新的内存空间，那意义就不大了。

取模的主要目的，是减少内存块的种类，以有限几个尺寸的内存块，应对绝大多数内存使用要求。

笔者看过 STL 的内存管理模块源代码，对其内存块的取模机制深表钦佩，因此，在笔者自己的内存池中，也是按照这种方式取模。

提示：32 位系统，有个字节对齐问题，即一个程序变量单元，如一个结构体，一个内存块，如果其尺寸不是 4Bytes 的整倍数，操作系统会按照比它大的整倍数分配内存，这其实也是操作系统在取模。比如我们的一个结构体为 7Bytes，操作系统分配时会分配 8Bytes，一个 14Bytes 的内存块，操作系统会分配 16Bytes，这主要是简化内存地址运算，以一定的内存消耗，来提升程序的运行速度。

我们对内存的取模也是这个原理，当然，我们不可能像操作系统那样，机械地以 4Bytes 为模数，那样，内存块种类还是太多，管理起来压力很大，内存池的效率也不高。

笔者仿造 STL 的取模方式，在内存池中按照如下逻辑取模，简单说来，就是从 16Bytes 开始，以两倍方式递增模数。直到 4G 内存为止。当然，实际使用时，超过 1M 的内存块，一般应用很少，即使有，基本上也属于应用程序永久缓冲区，很少会中途频繁释放，因此，笔者的内存池管理，一般模数为 16Bytes~1M 即可。

序号	内存块大小模数 (Bytes)	应对申请需求 (Bytes)
1	16	1~16
2	32	17~32
3	64	33~64
4	128	65~128
5	256	129~256
6	512	257~512
7	1k	513~1k
8	2k	1k+1~2k
9	4k	2k+1~4k
10	8k	4k+1~8k
11	16k	8k+1~16k
12	32k	16k+1~32k
13	64k	32k+1~64k
14	128k	64k+1~128k
15	256k	128k+1~256k
16	512k	256k+1~512k
17	1M	512k+1~1M

图 7.1：内存模数表

如表 7.1 所示，内存池实际上是一个树型数据结构在管理，每一种类型的内存块，构成一个链表，形成树的“右枝”，而所有右枝的链头，又是以一个链表在管理，形成树的“左枝”。请注意，这并不是二叉树，还是一颗普通的树结构。如下图所示：



图 7.2：内存管理树模型

我们举个例子，当应用程序申请一块 57Bytes 的内存块，程序逻辑会沿着树的左枝，从头到尾比对，首先是 16Bytes 的的管理链，由于 57>16，因此，无法在这根右枝进行管理，因此继续往下，32Bytes 也不行，64Bytes，比 57 要大，可以使用，因此，就在 64Bytes 这根右枝实现内存块管理。

管理原则：分配时，首先在合适的右枝寻找可用的内存块，如果有，则直接分配给应用程序重用该块，如果没有，则向系统申请一块 64Bytes 的内存块，分配给应用程序使用。而当应用程序释放时，内存块本身是 64Bytes 的，因此，可以直接挂回到 64Bytes 这根右枝，等待下次重用。

提示：这里面有一个隐含的推论，如果一个应用程序，需要一块 57Bytes 的内存块，那么，我们分配一块比它大的内存块，比如 64Bytes 的内存块，是完全可以的，应用程序不关心自己实际获得的内存块大小，同时，这种稍稍超大的分配机制，也不回引发任何的内存溢出 bug，反而更安全，因此，这种内存取模分配的思路，是完全可行的。

7.2.2 管理模型的优化

虽然上文我们讨论的是以链表方式管理，不过，在实做中，笔者发现一个问题，即链表效率不高，原因很简单，笔者的链表是以队列方式管理，每次从右枝取出内存块，是从链表头取出，但释放时，将内存块推回右枝，需要循环遍历到链表尾部进行挂链操作。这在高速的内存申请和释放时，会严重影响链表的效率。

笔者经过思考，发现一个问题，当一个内存块被推回一个右枝，其实已经是无属性的，比如，64Bytes 这个右枝上，挂的都是 64Bytes 大小的内存块，应用程序申请时，使用任何一块都是可以的，无需考虑这块是在链表头还是链表尾，同时，申请的内存块，都是需要初始化的，应用程序也不关心这块内存块是否刚刚被使用完，还是已经空闲很久了。笔者理解这个内存树的右枝，其实已经是前文所说的“被动池”逻辑了。

我们知道，在“推”入和“提取”这个逻辑上，“栈”的效率远高于“队列”，通常我们不使用栈的唯一原因，主要是栈是“后进先出”逻辑，而队列是“先进先出”逻辑，而我们常见的应用模型，一般都有数据顺序要求，因此，队列的使用场合，远多于栈结构。

但此处既然我们已经明确论证了，内存块无顺序需求，那么，我们完全可以使用栈模型来管理内存树的右枝，以提高效率。

这在实做时非常简单，当应用程序释放一块内存，我们需要推回右枝时，直接将其挂接到链表头即可，取消了无意义的循环遍历链表尾的操作，虽然，下次申请时，最后释放的一块内存会被最先分配使用，但这又有什么关系呢？

这个优化看似很小，但实做时威力惊人，经笔者测试，内存块的申请和释放吞吐量，在“队列”管理方式下，每秒仅 5 万次左右，一旦使用“栈”方式管理，迅速提升到 40~50 万次，提升了整整一个数量级。

正因为如此，笔者才将内存池最核心的内存管理模块，定名为内存栈（Memory Stack）。

提示：在进行程序开发时，很多时候，需要针对业务需求进行分析，实现针对性优化，很多时候，很小的一点优化，都可以大幅度提升程序的性能。反过来说，通用的优化其实不存在，只有深刻理解了业务需求之后，才有可能实施有效的优化方案。

提示：原则上，程序开发应该遵循“先实现，后优化”的原则，笔者常说的“先解决有无问题，再解决好坏问题”，也是这个意思，本章在此先讨论优化，是因为笔者这个内存池在实践中已经经过了多次优化，有条件讨论此事，并不意味着可以再程序实现前实施优化，请各位读者关注这个细节。事实上，本书展示的内存池，已经是笔者第 19 个版本，中间经过了十几次优化的结果。

7.2.3 关于链表管理的思考

讨论完上面的问题，我们再来讨论一下基本数据结构管理的问题。我们知道，虽然我们的内存池是树结构管理，但具体到每个右枝上，还是链表，而链表元素是动态申请的，依赖内部指向下一元素的指针，实现链接关系。

具体到我们内存块管理上，我们发现一个基本的链表元素，至少需要定义成如下形式：

```
typedef struct _CHAIN_TOKEN_  
{  
    struct _CHAIN_TOKEN_* m_pNext; //指向下一链表元素的指针  
    char* m_pBuffer;                //指向真实内存块的指针  
}SChainToken;
```

这就带来一个问题，链表的元素，应该分为两部分，一部分是实现链表管理的逻辑数据，如：`m_pNext`，另一部分，是业务相关的数据，如 `m_pBuffer`，这样的数据结构，造成程序开发非常麻烦。

比如一个简单的内存申请和释放动作，以上述数据结构管理，其基本逻辑如下：

内存申请：

- 1、检查链表有无空闲内存单元
- 2、如果有，提取其中的 `m_pBuffer`，准备返回给应用程序使用
- 3、从链表中卸载已经为空的链表管理元素，直接释放给系统（注意，无管控的内存块释放，内存碎片的隐患）

内存释放：

- 1、寻找合适的链，准备做挂链操作
- 2、申请一个链表管理单元，将内存块的指针放入其中的 `m_pBuffer`
- 3、执行挂链操作，填充 `m_pNext` 指针

大家注意到没有，我们本意是内存管理，减小内存碎片，但是，为了实现链表的管理，反而中间引入了一个多余的链表单元申请和释放逻辑，反而增加了内存碎片的产生可能，这种方法当然不可取。

另外，这里还有一个隐患，我们分配给应用程序的内存块，其中所有的内存单元，都是对应用程序透明的，应用程序可以任意使用，这说明，这块内存块中，没有存储任何关于内存块尺寸的信息，当应用程序释放指针时，我们面临一个问题，就是怎么确定这根指针指向的内存块，究竟有多大，应该挂在哪个右枝上，等待下次使用。

这个问题不解决，上述的内存释放逻辑的第一步，寻找合适的链，根本无法完成。

因此，为了记录内存块的尺寸信息，我们必须内部再建立一个映射表，将我们管理的每根内存块指针，在申请时的具体尺寸，都记录下来，等释放时，需要根据指针，逆查其对应的长度数据，才能完成功能。

笔者做开发有个原则：“简单的程序才是好程序”，上述逻辑虽然最终也能完成功能，但无论怎么看，都太复杂了，不是好的解决方案。

为此，笔者经过了较长时间的思考，发现所有问题的核心焦点，无非只有两条：

- 1、如何使链表的管理数据，不要发生新的动态内存分配。
- 2、如何使分配出去的指针，能够携带相关的缓冲区尺寸信息，避免额外的存储和查询压力。

经过分析，笔者突发奇想，既然我们内存池管理的就是内存块，就有存储能力，为什么我们不能利用内存块做一点自己的管理数据存储呢？大不了这个内存块实际可用内存，比我们从操作系统申请的，要小一点，但这又有什么关系呢？应用程序需要的只是自己需要的内存块，这个内存块原来有多大？能给应用程序使用的又有多大？应用程序并不关心。

经过考虑，笔者做了如下一个结构体：

```
typedef struct _TONY_MEM_BLOCK_HEAD_  
{  
    ULONG m_ulBlockSize;           //内存块的尺寸  
    struct _TONY_MEM_BLOCK_HEAD_* m_pNext; //指向下一链表元素的指针  
}STonyMemoryBlockHead;  
//本结构体的长度，经过计算，恒定为 8Bytes  
const ULONG STonyMemoryBlockHeadSize=sizeof(STonyMemoryBlockHead);
```

上述结构体，包含了内存块尺寸信息，来满足释放时查找的需求，同时，包含了指向下一元素的指针，这个指针，在分配给应用程序使用时，是无效的，只有当这个内存块挂在链表中时，才有意义。

笔者这么思考，当我们向系统申请一个内存块，比如说 64Bytes，我们内存池占用其中最开始的 8Bytes 来存储上述信息，也就是说，实际能给应用程序使用的，只有 56Bytes。如图 7.3：

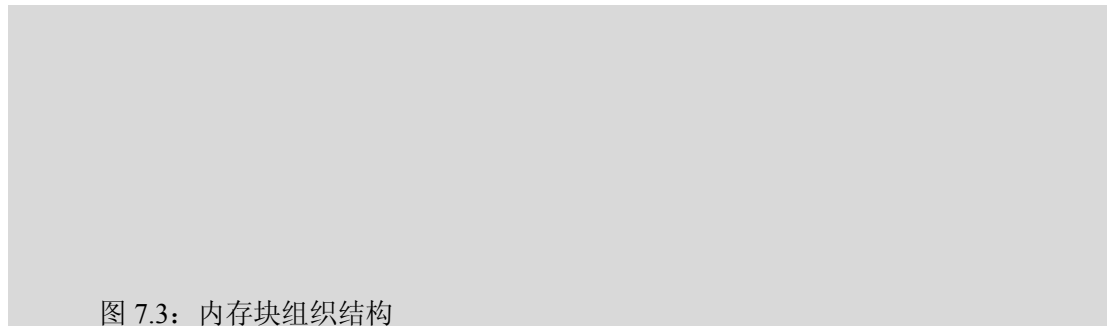


图 7.3：内存块组织结构

我们假定这个内存块的真实尺寸为 N Bytes，我们从系统申请的首指针为 p0，那么，我们占用 8Bytes 作为管理使用，当应用程序申请时，我们真实分配给应用程序的指针为 p1=(p0+8)。这样，当应用程序释放时，我们只需要执行 p0=(p1-8)，即可求出原始首地址，并以此获得所有的管理信息。当最后向系统释放内存时，我们只要记得释放 p0 即可。

提示：此处可能出于业务考虑，有点违背 C 和 C++ 无错化程序设计方法中，关于指针不得参与四则运算的原则，不过，没办法，需求如此，只有这条路走了。因此，违背就违背一点了。

唯一需要我们注意的细节，是我们在分析应用程序的内存申请需求时，不能以申请的内存块的真实尺寸进行比对，而应该比对减去 8Bytes 之后的数据。即 64Bytes 这个右枝上提供的内存，只有 56Bytes 大小，如果超过这个值，请找下一链，即到 128 Bytes 这个右枝处理，当然，此时的 128 Byte 的右枝，也仅能提供 120 Bytes 的内存块，以此类推。

有鉴于此，笔者做了如下的宏定义，来界定所有的计算行为：

```

//根据一个应用程序数据块的长度，计算一个内存块的真实大小，即 n+8
#define TONY_MEM_BLOCK_SIZE(nDataLength) \
    (nDataLength+STonyMemoryBlockHeadSize)
//根据向系统申请的内存块，计算其应用程序数据内存的真实大小，即 n-8
#define TONY_MEM_BLOCK_DATA_SIZE(nBlockSize) \
    (nBlockSize-STonyMemoryBlockHeadSize)
//根据应用程序释放的指针，逆求真实的内存块指针，即 p0=p1-8
#define TONY_MEM_BLOCK_HEAD(pData) \
    ((STonyMemoryBlockHead*)((char*)pData)-STonyMemoryBlockHeadSize)
//根据一个内存块的真实指针，求数据内存块的指针，即 p1=p0+8
#define TONY_MEM_BLOCK_DATA(pHead) \
    (((char*)pHead)+STonyMemoryBlockHeadSize)
//最小内存块长度，16 Bytes，由于我们管理占用 8 Bytes，这个最小长度不能再小了，
//否则无意义，即使这样，我们最小的内存块，能分配给应用程序使用的，仅有 8 Bytes。
#define TONY_XIAO_MEMORY_STACK_BLOCK_MIN 16
//这是管理的最大内存块长度，1M，如前文表中所示，超过此限制，内存池停止服务
//改为直接向系统申请和释放。
#define TONY_XIAO_MEMORY_STACK_MAX_SAVE_BLOCK_SIZE (1*1024*1024)

```

7.2.4 内存块元素

笔者经过规划，内存栈分为两层实施管理，一层是内存块管理单元，每个单元管理一根右枝，即一种尺寸的内存块链表，而第二层则是内存栈，以链表形式管理内存管理单元，实现左枝，本小节向大家介绍内存块元素的实现。

7.2.4.1 类声明

内存块管理单元，本质上还是一个 C++ 的类，只有最底层的内存块，是以纯 C 的结构体在进行管理，主要的目的是不希望引发新的对象 new 行为，导入不必要的，以及不受控的动态内存申请和释放。


```

//内存栈最基本的管理单元，笔者习惯以 Token 定名单位，单元，元素等结构，仅仅是习惯而已。
//本类使用了一定的递归逻辑
class CTonyMemoryStackToken
{
public:
    //构造函数和析构函数，注意，此处要求输入本内存块管理的基本内存大小，如 64 Bytes
    //以及 Debug 对象的指针，本模块需要利用 Debug 进行中间的输出。
    CTonyMemoryStackToken(int nBlockSize,CTonyLowDebug* pDebug);
    ~CTonyMemoryStackToken();

public:
    //基本的申请函数 Malloc，返回成功申请的内存块指针，注意，是 p1，不是 p0
    //注意其中的统计设计，这是为了 Debug 和观察方便
    //另外，请注意，统计变量是传址操作，因此，可以在函数内部修改，回传信息
    // nAllBlockCount- nMemoryUse=内部的空闲内存块数量
    void* Malloc(int nSize,          //应用程序申请的内存尺寸
        CMRSWint& nAllBlockCount,    //统计变量，返回管理过的内存块总数
        CMRSWint& nMemoryUse);       //统计变量，返回被应用程序使用的内存块数量
    //释放函数，此处的 bCloseFlag 是一个特殊优化，即在程序退出时
    //释放的内存不会再返回内存栈，直接被 free 掉，以加快程序退出速度。
    //注意，此处应用程序传进来的是 p1，内部需要逆向计算出 p0 实现操作。
    bool Free(void* pPoint,bool bCloseFlag);
    //这是纯粹的性能观察函数，这也是笔者开发底层模块的一个习惯
    //底层模块，由于很少有 UI 交互的机会，因此，如果内部有什么问题，程序员很难观察到
    //由此可能导致 bug，因此，一般对底层模块，笔者习惯性做一个打印函数
    //打印一些特殊信息，帮助程序员观察性能或 bug。
    void PrintStack(void);

private:
    //系统退出时，递归销毁所有内存块的函数。
    //笔者一般习惯以 Brother（兄弟）或 Next（下一个）来定名左枝，
    //以 Son（儿子）来定名右枝
    void DestroySon(STonyMemoryBlockHead* pSon);

private:
    //第一个儿子的指针，这就是链头了
    STonyMemoryBlockHead* m_pFirstSon;
    //这是指向兄弟节点，即左枝下一节点的指针
    CTonyMemoryStackToken* m_pNext;
    //最重要的设计，线程安全锁，注意，为了提升效率，这里使用了单写多读锁
    //根据本类定义，锁仅保护上述两个指针变量
    CMultiReadSingleWriteLock m_Lock;
    //内部保存一下本对象管理的内存块尺寸，就是构造函数传入的数据
    //供比对使用
    ULONG m_ulBlockSize;
    //性能统计变量，请注意，这些都是单写多读锁安全变量，本身是现成安全的。
    CMRSWint m_nBlockOutSide;        //分配出去的内存块数量
    CMRSWint m_nBlockInSide;         //内部保留的空闲内存块数量
    CTonyLowDebug* m_pDebug;         //debug 对象指针
};

```

这里有几个特点请大家关注：

1、笔者在程序中使用了递归，虽然 C 和 C++ 无错化程序设计方法，强调尽量减少使用递归，但无可否认，递归能大大简化程序的实现。因此，在能基本确定递归层数的应用场合，还是可以使用一点递归的。根据经验，动态内存块的申请和释放，一般都是在几十，几百的数量级，此时，递归层数有限，属于可控范畴，笔者于是使用了递归简化程序设计。

2、请注意单写多读锁的使用，由于这种链表遍历比较，很多时候以读为主，因此，使用单写多读锁，可以大大加快程序的效率。

3、请注意统计变量的设计，debug 对象的导入，以及 PrintfStack 函数的设计，底层模块，很少有 UI 交互机会，出了 bug 不容易被观察，程序员应该有意识地主动设计一些性能统计，Debug 辅助函数和变量，帮助后期的调试工作。

7.2.4.2 构造函数和析构函数

内存栈的构造函数和析构函数相对比较简单，其中，析构函数特别做了很多递归删除的逻辑，在退出时，摧毁所有的右枝（儿子）和左枝（兄弟）。以此确保内存不泄露。

```
CTonyMemoryStackToken::CTonyMemoryStackToken(int nBlockSize,
    CTonyLowDebug* pDebug)
{
    m_pDebug=pDebug;                //保存 Debug 对象指针
    m_ulBlockSize=(ULONG)nBlockSize; //保存本对象管理的内存块尺寸
    m_nBlockOutSide.Set(0);          //统计变量初始化
    m_nBlockInSide.Set(0);
    m_Lock.EnableWrite();            //注意单写多读锁的使用，这是写锁
    {
        m_pFirstSon=null;           //指针变量初始化
        m_pNext=null;               //注意，这是在锁保护下进行的
    }
    m_Lock.DisableWrite();
}

CTonyMemoryStackToken::~~CTonyMemoryStackToken()
{
    if(m_nBlockOutSide.Get())
    {
        //注意此处，当检测到还有内存块在外被应用程序使用，
        //而本对象又需要析构，即内存池停止服务时，报警提醒程序员。
        TONY_DEBUG("Tony Memory Stack: lost %d * %d\n",
            m_ulBlockSize,m_nBlockOutSide.Get());
    }
    m_Lock.EnableWrite();
    {
        //此处，在锁保护下，摧毁所有的右枝和左枝，这是递归运算
        if(m_pFirstSon) DestroySon(m_pFirstSon);
        m_pFirstSon=null;
        if(m_pNext) delete m_pNext;
        m_pNext=null;
    }
    m_Lock.DisableWrite();
}
```

这里面有几个细节请大家关注：

从上文我们可以看出，虽然我们右枝的内存块，使用的是纯 C 的结构体管理，并且，严格按照内存池管理原则，运行期间只申请，不释放，确保无内存碎片，但是，左枝的管理单元，还是 C++ 对象构成的链表，这里存在一个动态创建对象的问题。

不过，我们前期已经分析过，由于内存块取模，因此，左枝的总数有限，只有 17 个链表节点，在运行期间也是只申请不释放，因此可以认为不会造成内存碎片。

另外，在析构函数中，我们检测到还有内存块分配在外，仅仅报警，没有替应用程序释放，这是因为由于我们前面的优化，链表节点管理内容和实际的内存块是一个整体，分配时，一起被分配出去，本地并未保存指针，因此，无法释放。

不过，由于内存栈本身的生命，就是贯穿于整个程序运行生命周期，最后退出时，一般应用程序也结束了，即使有内存泄漏，也不重要了，因为随着进程的结束，整个进程内存地址空间被操作系统摧毁，远堆已经不存在，也就无所谓内存泄漏了。

当然，此处的报警比较简陋，仅仅能报出有多少内存块忘了释放，具体的内存块属于哪个模块使用，我们放在下一节的注册单元中实现，此处不再赘述。

7.2.4.3 Malloc

Malloc，是内存池最重要的功能，在实际使用中，需要完全替代 C 语言的 malloc，和 C++ 语言 new 的内存分配功能，我们在开发时，至少要实现如下功能：

1、线程安全

2、检索本模块管理的内存块，能否满足内存申请需求，如果能满足，则

a) 检索右枝，有无空闲内存块，有则提取出来，返回给应用程序使用，同时，修改相应的统计数据。

b) 如果右枝为空，则使用 malloc，向操作系统申请内存块，此时，申请的内存块不入右枝，直接返回给应用程序使用。

3、如果申请内存比本模块管理的内存块大，则

a) 检索左枝，有无兄弟节点，无则创建兄弟节点，兄弟节点管理的内存块是本对象的两倍大小。

b) 将申请传递给左枝兄弟节点处理。

4、如果申请的内存块，超过最大限额 1M，则直接向操作系统申请，本内存池不做处理。

代码实现如下：

```

//Malloc 逻辑
void* CTonyMemoryStackToken::Malloc(int nSize,
    CMRSWint& nAllBlockCount,
    CMRSWint& nMemoryUse)
{
    void* pRet=null;           //准备返回的指针
    STonyMemoryBlockHead* pNew=null; //中间变量
    //请注意这个宏的使用
    //根据一个应用程序数据块的长度，计算一个内存块的真实大小，即 n+8
    //define TONY_MEM_BLOCK_SIZE(nDataLength) \
    //    (nDataLength+STonyMemoryBlockHeadSize)
    //这是申请大小+8，再和本对象管理大小比较，以此修订管理数据带来的偏差
    //这里还有一个技巧，m_ulBlockSize 没有使用锁保护，这是因为 m_ulBlockSize
    //从构造函数填充后，所有的使用都是只读的，不再需要锁保护，以简化锁操作
    if((TONY_MEM_BLOCK_SIZE(nSize))<m_ulBlockSize)
    {
        //这表示本对象的大小可以满足申请需求，申请将从本节点右枝完成
        m_Lock.EnableWrite(); //加锁
        {
            //判断是否有空闲的内存块备用
            if(!m_pFirstSon)
            {
                //如果没有，需要向系统新申请一个内存块
                //注意这里的强制指针类型转换，申请的内存块，在本函数直接
                //转换成管理结构体指针，以便操作
                //注意申请的大小就是 m_ulBlockSize，因此
                //本对象申请的所有内存块，都是一样大小的。
                pNew=(STonyMemoryBlockHead*)malloc(m_ulBlockSize);
                if(pNew) //安全检查
                {
                    //统计函数+1，因为这个内存块马上会分配出去，
                    //因此，修订 m_nBlockOutSide+1, InSide 不加。
                    m_nBlockOutSide.Add();
                    //这是帮助上层统计总内存字节数，注意，这是传址调用，数据会上传
                    nMemoryUse.Add(m_ulBlockSize);
                    //初始化新申请的内存块，填充大小信息
                    //注意，下面的 Free 要用到这个信息查找对应的左枝节点。
                    pNew->m_ulBlockSize=m_ulBlockSize;
                    //由于链表指针只有在管理时使用，分配出去的，暂时清空
                    pNew->m_pNext=null;
                    //这里最关键，请注意，使用了宏计算，p1=p0+8，求得 p1，
                    //返回给应用程序使用
                    pRet=TONY_MEM_BLOCK_DATA(pNew);
                    //统计变量，内存块总数+1
                    nAllBlockCount.Add();
                }
            }
            //如果申请失败，则直接返回 null 给应用程序
        }
    }
}

```

```

else
{
    //这是有空闲内存块的情况
    //直接提取链表第一块，也就是栈上最新加入的内存块
    pNew=m_pFirstSon;
    //这是指针修订，注意，本类中 m_pFirstSon 已经指向了原来第二块内存
    m_pFirstSon=pNew->m_pNext;
    //同上，分配出去的内存块，m_pNext 无用，清空
    pNew->m_pNext=null;
    //求出 p1=p0+8，返回给应用程序使用
    pRet=TONY_MEM_BLOCK_DATA(pNew);
    //注意此处，这是把内部的内存块再分配重用
    //因此，OutSide+1，InSide-1
    m_nBlockOutSide.Add();
    m_nBlockInSide.Dec();
}
}
m_Lock.DisableWrite();    //解锁
}

```

```

else
{
    m_Lock.EnableWrite();    //加写锁
    {
        //这是检测兄弟节点是否已经创建，如果没有，则创建之。
        //注意，此处对 m_pNext 有写关系，因此，加写锁。
        if(!m_pNext)
        {
            m_pNext=new CTonyMemoryStackToken(
                m_ulBlockSize*2,m_pDebug);
        }
    }
    m_Lock.DisableWrite();    //解写锁
    //此处锁读写模式转换非常重要，下面专门论述
    m_Lock.AddRead();        //加读锁
    {
        //将请求传递给兄弟节点的 Malloc 函数处理
        if(m_pNext)
            pRet=m_pNext->Malloc(nSize,nAllBlockCount,nMemoryUse);
        //如果兄弟节点创建失败，表示系统内存分配也失败，返回空指针给应用程序
    }
    m_Lock.DecRead();        //解读锁
}
return pRet;
}
}

```

这段 Malloc 函数，逻辑上还是比较明确，基本实现了我们的需求，不过，其中可能各位读者最为费解的，就是后半段的读写锁转化过程。笔者这里尝试给大家解说一下：

1、这个内存池，目的是应对应用程序各种动态内存申请需求，因此，其需求一定是跨线程的，也就是说，很可能同一时间，Malloc 这个函数，有几个线程在 call，其内部的单元，一定要注意锁保护。

2、为了提高效率，笔者在本类中使用了单写多读锁来解决问题，这就要求程序书写时，必须很清楚对需要保护的内存变量，当前是读操作还是写操作。

3、我们知道，本对象是内存树的左枝节点，同时，管理着整整一个右枝的内存块的分配工作，因此，针对右枝的所有操作，我们视为读写并发，在本函数的前半节，即需求由本对象处理时，我们全部加写锁保护。

4、当需求大于本对象的内存大小，我们需要把需求传递给兄弟节点处理时，此时要细分，我们的 `m_pNext` 有两种操作需求：

a) 当 `m_pNext` 为空，我们需要 `new` 一个兄弟节点时，此时对 `m_pNext` 是写操作，必须加写锁。

b) 当 `m_pNext` 不为空，我们把需求传递给兄弟节点的 `Malloc` 函数处理，其实，我们此时仅仅是读取 `m_pNext` 的值，是读操作，此时，如果继续加写锁，虽然安全，但显得效率太低，需要改成读锁。

举个例子，我们假定两个线程，同时发起对内存池的申请动作，一个申请 64Bytes，另一个申请 32Bytes，如果上文遍历操作时，针对 `m_pNext` 是具有排他性的写锁，则在递归穿越 16Bytes 的左枝单元上，上述两个递归遍历请求被强行同步，只有一个请求能通过，另一个被阻塞。

也就是说，如果我们在递归时采用写锁，则内存池同一时间，只能响应一个操作，效率极低，而改为读锁，由于读锁允许并发，则上述两个动作，会顺利通过 16Bytes 的这个单元的管理，直接递归到合适的单元完成，由于上述两个动作又是不同大小的内存需求，由不同的内存管理单元完成操作，因此，可以认为没有锁冲突，是完全并发的。

这个修改非常重要，经过实测，仅此一处修改，内存池的分配效率就从每秒几千次，扩展到几十万次，提升效果异常明显，因此，请各位读者务必关注这个要点。

同时，单写多读锁的威力，由此也可见一斑。

7.2.4.4 Free

与 `Malloc` 相对，`Free` 也是内存池最重要的功能之一，它至少应该实现如下逻辑：

1、释放的内存块如果太大，超出了 1M 的限制，则本内存池不做重用管理，直接调用 C 语言的 `free`，释放给系统。

2、判断释放的内存块尺寸是否归本节点管理，如果在管理范围内，则利用宏计算，逆求 `p0=p1-8`，获得原始内存块指针，并且检索管理结构体内部，内存块的大小，并递归到合适的管理单元处理，推入其中的右枝栈，等待下次重用。

3、如果不是本节点管理，则寻找兄弟节点，将释放请求交付兄弟节点的 `Free` 函数处理。

4、如果发现 `bCloseFlag` 为真，表示系统开始退出流程，则释放的内存块不再推入栈，直接向系统释放掉。

5、中间注意修订统计变量，所有的操作需要线程安全。

经过考虑，笔者把这段函数写成如下构型：

```
//Free函数，允许传入 bCloseFlag 标志，注意，外部传入的指针为 p1，需要逆求 p0=p1-8  
//在某种情况下，Free 可能会失败，因此，需要返回一个 bool 值，表示释放结果。  
bool CTonyMemoryStackToken::Free(void* pPoint,bool bCloseFlag)  
{  
    bool bRet=false; //准备返回值变量  
    //注意这里，这个 pOld，已经是计算的 p0=p1-8  
    STonyMemoryBlockHead* pOld=TONY_MEM_BLOCK_HEAD(pPoint);  
    //首先检测指定内存块大小是否由本对象管理。  
    if(m_ulBlockSize==pOld->m_ulBlockSize)  
    {  
        //此处是判断内存块对象是否超限，如果超出限额，直接释放，不做重用处理。  
        //因此，对于超限的内存块，其管理节点对象总是存在的，但对象的右枝为空。  
        if(TONY_XIAO_MEMORY_STACK_MAX_SAVE_BLOCK_SIZE<=m_ulBlockSize)  
        {  
            free(pOld);  
            m_nBlockOutSide.Dec(); //注意，这里修订 OutSide 统计变量。  
        }  
        else if(bCloseFlag)  
        { //当 CloseFlag 为真，不再推入右枝，直接释放  
            free(pOld);  
            m_nBlockOutSide.Dec();  
        }  
        else  
        { //当所有条件不满足，正常推入右栈，准备重用内存块  
            m_Lock.EnableWrite(); //加写锁  
            { //请注意这里，典型的压栈操作，新加入内存块，放在第一个，  
                //原有的内存块，被链接到新内存块的下一个，先进先出  
                pOld->m_pNext=m_pFirstSon;  
                m_pFirstSon=pOld;  
            }  
            m_Lock.DisableWrite(); //解除写锁  
            m_nBlockOutSide.Dec(); //修订统计变量  
            m_nBlockInSide.Add();  
        }  
        //请注意返回值的位置，只要是本节点处理，都会返回 true。  
        bRet=true;  
    }  
    else  
    { //这里有一个默认推定，请大家关注：  
        //当一个程序，所有的动态内存申请都是由本内存池管理是，  
        //内存块一定是被本内存池分配，因此，当内存块释放时，  
        //其对应的左枝节点，即内存管理单元节点，一定存在  
        //因此，Free 不再处理 m_pNext 的创作问题，而是直接调用  
        m_Lock.AddRead(); //加读锁，原因前面已经论述  
        { //此处递归操作  
            if(m_pNext)  
                bRet=m_pNext->Free(pPoint,bCloseFlag);  
        }  
        m_Lock.DecRead(); //解读锁  
    }  
    return bRet;  
}
```

7.2.4.5 DestroySon

DestroySon 是对象析构时，帮助内存块管理单元，清空右枝的函数，考虑到在某些特殊场合，如大并发量的网络服务系统，动态内存块较多，很可能导致右枝很长，因此，针对这个函数，特地做了优化，不使用递归方式，而使用 while 循环。各位读者有兴趣，也可以研究一下递归和循环两种链表控制方式的差异性。

```
//摧毁所有的子节点
void CTonyMemoryStackToken::DestroySon(STonyMemoryBlockHead* pSon)
{
    //本函数是典型的对象内私有函数，被其他线程安全的函数调用，因此，内部不加锁。
    STonyMemoryBlockHead* pObjNow=pSon;    //中间变量
    STonyMemoryBlockHead* pOnjNext=null;
    while(1)                                //循环方式
    {
        if(!pObjNow) break;                //循环跳出点
        pOnjNext=pObjNow->m_pNext;
        free(pObjNow);
        m_nBlockInSide.Dec();
        pObjNow=pOnjNext;
    }
}
```

7.2.4.6 PrintInfo 相关函数

前文我们讲过，传输类代码，一般都运行于系统底层内核之中，很少有直面用户的机会。因此，其运行信息相对封闭，像一个黑盒子一样，运行时很难调试和观察性能。

这就要求商用数据传输工程的程序员，要有很强的自我调试意识，在程序开发之初，就应该有意识地在程序中添加一些 debug 以及性能观察代码，帮助后期的工作。

本类中的 m_nBlockOutSide, m_nBlockInSide 这两个变量的设置，就是为了这个目的，同样，笔者也习惯在很多底层工具类中，添加一个公有方法函数，PrintfInfo，就是为了调试观察时，向屏幕输出一些关键信息，帮助程序员理解，这在后续很多工具类中，大家都可以看到。

本类的观察函数，由于是栈式管理，笔者定名为 PrintStack。

```
//内存管理单元的信息输出函数
void CTonyMemoryStackToken::PrintStack(void)
{
    //这是一个典型的递归函数，一次打印左枝上所有的内存管理单元的内容
    if(m_nBlockInSide.Get()+m_nBlockOutSide.Get())
    {
        //有值则打印，无值不输出
        TONY_XIAO_PRINTF(" [%ld] stack: all=%d, out=%d, in=%d\n",
            m_ulBlockSize, //这是提示内存块的大小
            m_nBlockInSide.Get()+m_nBlockOutSide.Get(), //这是所有内存块的总数
            m_nBlockOutSide.Get(), //分配出去的内存块
            m_nBlockInSide.Get()); //内部保存备用的内存块
    }
    m_Lock.AddRead(); //加读锁
    if(m_pNext)
    {
        m_pNext->PrintStack(); //注意，这里递归
    }
    m_Lock.DecRead(); //解读锁
}
```


7.2.5 内存栈

前一小节，我们介绍了内存块的管理单元，该类可以有效组织内存树的右枝，当然，我们还需要一个类来聚合这个内存块管理单元类，实现左枝的管理，最终，以树的形式，构建内存管理逻辑。

由于前文讲过，我们的核心管理逻辑是“栈”，因此，这个对外公开的内存管理模块，笔者把它定名为“内存栈”（MemoryStack）。

7.2.5.1 类声明

内存栈的类声明比较简单，关键是相对于右枝的内存块管理单元，增加了一个 ReMalloc 函数，即实现内存块的大小重定义，这是本小节的重点。

```
//内存栈
class CTonyMemoryStack
{
public:
    CTonyMemoryStack(CTonyLowDebug* pDebug);    //构造函数，需要传入 Debug 对象
    ~CTonyMemoryStack();
public:
    //重定义指针指向的内存单元地址空间大小，给出旧指针，和新的空间尺寸
    //如果成功，返回有效的新指针，指向重定义大小之后的新空间，失败返回 null
    //注意，本函数可以把旧空间的数据内容备份到新空间，但这不一定，取决与新空间大小
    void* ReMalloc(void* pPoint,                //旧指针
        int nNewSize,                          //新的尺寸需求
        bool bCopyOldDataFlag=true);           //是否备份旧数据的标志（默认“真”）
    //Malloc 和 Free 函数，调用内存块管理单元方法完成
    void* Malloc(int nSize);
    bool Free(void* pPoint);
    //内部信息输出函数，下文会讲到其差别
    void PrintStack(void);
    void PrintInfo(void);
    //作为一项优化，如果 app 设置这个关闭标志，
    //所有的 free 动作都直接调用系统的 free 完成，不再保留到 stack 中。
    //加快退出速度。
    void SetCloseFlag(bool bFlag=true);
    CTonyLowDebug* m_pDebug;                    //debug 对象，此处为了优化公开
private:
    CTonyMemoryStackToken* m_pHead;             //注意，这是左枝开始的第一个节点
    CMRSWint m_pMaxPoint;                       //统计当前用到的最大内存指针
    CMRSWint m_nAllBlockCount;                  //统计所有在用的内存块
    CMRSWint m_nMemoryUse;                     //统计内存使用的总字节数
    CMRSWbool m_CloseFlag;                     //关闭标示
};
```

请注意，这里面提出了很重要的几个统计概念：

- 1、m_pMaxPoint，这通常表示内存池有史以来得到的最大指针，这可以帮助观察应用程序使用的内存是不是过大，如果这个值一直在增加，则表示可能有内存泄漏。
- 2、m_nAllBlockCount，内存池有史以来，分配的总内存块大小，由于内存块的重用特性，这个数值远小于程序实际使用的内存块数，这也是用来帮助检查内存是否泄漏的工具。
- 3、m_nMemoryUse，统计以字节为单位，内存实际使用情况，供程序最大使用内存分析，帮助书写用户说明，建议用户准备多大内存的计算机等，也能帮助观察内存是否泄漏。

7.2.5.2 构造函数和析构函数等工具函数

构造函数和析构函数主要完成初始化动作以及最后的收尾工作，避免出现 bug。

```
//构造函数
CTonyMemoryStack::CTonyMemoryStack(CTonyLowDebug* pDebug)
{
    m_CloseFlag.Set(false);                //关闭标志清空
    m_pDebug=pDebug;                        //保存 debug 对象指针
    m_pMaxPoint.Set(0);                     //统计变量赋初值
    m_nAllBlockCount.Set(0);
    m_nMemoryUse.Set(0);
    m_pHead=new CTonyMemoryStackToken(     //以最小内存块尺寸 16Bytes,
        TONY_XIAO_MEMORY_STACK_BLOCK_MIN, //构造左枝第一个节点
        m_pDebug);
    //注意，在整个运行期间，左枝节点只创建，不摧毁，因此，不会因为
    //动态对象而造成内存碎片
}
//析构函数
CTonyMemoryStack::~CTonyMemoryStack()
{
    //退出时，汇报一下使用的最大指针，方便程序员观察
    TONY_DEBUG("Memory Stack: Max Point= 0x%p\n",m_pMaxPoint.Get());
    //清除左枝，这是递归删除，由每个节点的析构完成
    if(m_pHead)
    {
        delete m_pHead;
        m_pHead=null;
    }
}
//公有方法，设置 Close 标志
void CTonyMemoryStack::SetCloseFlag(bool bFlag)
{
    m_CloseFlag.Set(true);
}
```

7.2.5.3 Malloc 和 Free

Malloc 和 Free，主要调用上一小节，内存块管理单元的公有方法来完成。

```

void* CTonyMemoryStack::Malloc(int nSize)
{
    void* pRet=null;
    if(0>=nSize)          //防御性设计，申请长度<=0 的内存空间无意义
    {
        TONY_DEBUG("CTonyMemoryStack::Malloc(): ERROR! nSize=%d\n",nSize);
        return pRet;
    }
    if(m_pHead)
    {
        //注意，此处进入递归分配，上一小节逻辑，递归到合适的管理单元分配
        pRet=m_pHead->Malloc(nSize,m_nAllBlockCount,m_nMemoryUse);
        //统计功能，统计最大的指针
        if(m_pMaxPoint.Get()<(int)pRet)
            m_pMaxPoint.Set((int)pRet);
    }
    return pRet;
}
bool CTonyMemoryStack::Free(void* pPoint)
{
    bool bRet=false;
    if(m_pHead)          //递归调用左枝节点上的 Free，完成 Free 功能
    {
        bRet=m_pHead->Free(pPoint,m_CloseFlag.Get());
    }
    return bRet;
}

```

7.2.5.4 ReMalloc

ReMalloc 是内存栈很重要的一个功能，在商用传输工程的很多场合，我们会使用接收 Buffer，而这个 Buffer 的大小，很可能会因为接收的报文不同，而不断调整大小，如果内存池不提供 ReMalloc 功能，应用程序自己来实现，则会使应用程序变得很繁琐，由此也可能引发不必要的 bug。

因此，笔者把这个方法抽象为一个通用的内存栈方法，帮助应用程序减少 bug。

```

//ReMalloc 功能，改变一个指针指向的内存区的尺寸。
//假定该指针是本内存池分配的，因此，根据给出的指针 p1，可以逆推算出 p0=p1-8，
//并由此获得原有尺寸的真实大小。
void* CTonyMemoryStack::ReMalloc(void* pPoint,          //原有指针, p1
    int nNewSize,          //新尺寸
    bool bCopyOldDataFlag) //拷贝旧数据标志
{
    void* pRet=null;
    STonyMemoryBlockHead* pOldToken=null;          //旧的 p0
    int nOldLen=0;          //旧的长度
    if(0>=nNewSize)          //防御性设计，防止申请非法的长度 (<=0)
    {
        //请注意打印格式，以函数名开始，方便程序员观察 Debug 信息时定位
        TONY_DEBUG("CTonyMemoryStack::ReMalloc(): ERROR! nNewSize=%d\n",
            nNewSize);
        goto CTonyMemoryStack_ReMalloc_Free_Old;
    }
    //请注意这里，调用宏计算 p0=p1-8
    pOldToken=TONY_MEM_BLOCK_HEAD(pPoint);
    //求的原内存块的大小
    nOldLen=pOldToken->m_ulBlockSize;
    //比较新的长度和内存块总长度的大小，
    //特别注意，NewSize 被宏计算+8，求的是内存块大小，这是修订误差
    if (TONY_MEM_BLOCK_SIZE(nNewSize)<= (ULONG)nOldLen)
    {
        //我们知道，内存栈管理的内存块都是已经取模的内存块，
        //其长度通常都是 16Bytes 的整倍数，并且比应用程序申请的内存要大
        //因此，很可能应用程序新申请的内存大小，并没有超过原内存块本身的大小
        //此时，就可以直接将原内存块返回继续使用，不会因此导致内存溢出等 bug
        //比如，应用程序第一次申请一个 260Bytes 的内存块，根据内存取模原则
        //内存池会分配一个 512Bytes 的内存块给其使用，有效使用空间 512-8=504Bytes
        //而当第二次，应用程序希望使用一个 300Bytes 的内存区，还是小于 504Bytes
        //此时，ReMalloc 会返回旧指针，让其继续使用，而不是重新申请，以提高效率。
        //另外，从这个逻辑我们也可以得知，如果调整的新尺寸本来就比原空间小
        //比如第二次调整的尺寸不是 300Bytes，而是 100Bytes，
        //也会因为这个原因，继续使用原内存，避免二次分配
        //这是一个典型的空间换时间优化，以一点内存的浪费，避免二次分配，提升效率
        //由于这是原内存返回，自然保留原有数据，本函数无需拷贝数据
        pRet=pPoint;
        //请注意这里，跳过了第一跳转点，直接跳到正常结束
        //即传入的指针不会被 Free，而是二次使用。
        goto CTonyMemoryStack_ReMalloc_End_Process;
    }
    //当确定新的尺寸比较大，原有内存无法承受，则必须调用内存管理单元，重新申请一块
    pRet=m_pHead->Malloc(nNewSize,m_nAllBlockCount,m_nMemoryUse);
    //当新内存申请成功，根据拷贝表示拷贝数据
    if((pRet)&&(pPoint))
        if(bCopyOldDataFlag)
            memcpy(pRet,pPoint,nOldLen);
CTonyMemoryStack_ReMalloc_Free_Old:
    //第一跳转点，出错时，或正确结束时，均 Free 旧指针
    m_pHead->Free(pPoint,m_CloseFlag.Get());
CTonyMemoryStack_ReMalloc_End_Process:
    //正常退出跳转点，返回新指针
    return pRet;
}

```

7.2.5.5 PrintfInfo 相关函数

由于内存栈内部实际上管理的是一颗树，内部数据结构相对比较复杂，如何在运行期观察内存栈的使用情况，就变得比较麻烦。

笔者经过思考，以及开发中的测试，为内存栈准备了两个 `PrintfInfo` 函数，一个用来简单打印几个关键变量，适用于少量输出场合，另外一个则详细输出目前内存树的详细情况，方便仔细观察。

```
//打印关键变量的值
void CTonyMemoryStack::PrintInfo(void)
{
    TONY_XIAO_PRINTF("block=%d, use=%d kbytes, biggest=%p\n",
        m_nAllBlockCount.Get(),           //所有内存块的总数
        m_nMemoryUse.Get()/1024,          //总内存使用大小 (KBytes)
        m_pMaxPoint.Get());               //最大指针
}
//递归调用内存管理单元的对应方法，打印整个树的情况。
void CTonyMemoryStack::PrintStack(void)
{
    if(m_pHead)
        m_pHead->PrintStack();
}
```

下面是运行时输出情况：

```
[Mon Jun 29 09:30:00 2009] *****
//这一行是 CTonyMemoryStack::PrintInfo 打印出来的信息
//表示当前使用了 16 个内存块，共计使用了 1025k 内存（1M 多一点），最大的指针是 0x00B50048
[Mon Jun 29 09:30:00 2009] block=16, use=1025 kbytes, biggest=00B50048
//这是 CTonyMemoryStack::PrintStack 打印的信息
//16 字节的内存块，共计用了 1 块，不过，目前没有使用，在内存树上备用。
[Mon Jun 29 09:30:00 2009] [16] stack: all=1, out=0, in=1
//32 字节的内存块，共计 9 块，其中，5 块正在使用，4 块备用，下面以此类推
[Mon Jun 29 09:30:00 2009] [32] stack: all=9, out=5, in=4
[Mon Jun 29 09:30:00 2009] [128] stack: all=1, out=1, in=0
[Mon Jun 29 09:30:00 2009] [256] stack: all=4, out=4, in=0
[Mon Jun 29 09:30:00 2009] [1048576] stack: all=1, out=1, in=0
[Mon Jun 29 09:30:00 2009] *****
```

大家可以看到，由于这两个函数的作用，内存池虽然是一个运行在最底层的函数，但我们程序员仍然可以很清晰地看到其运行状况，调试起来十分方便。

这里给大家提示一些笔者进行性能观察的要点：

- 1、所有的数字，不能疯长。即中间过程可以动态变化，但要有涨有跌，不能一味上涨，否则，程序一定有 bug，很可能是内存泄漏，软件会在将来某个时刻挂死。
- 2、内存块的总量，在运行一段时间后，应该趋于不变，如果系统运行几个小时，内存块仍然在持续缓慢上涨，则有可能程序有内存泄漏的 bug，只是泄漏很少，不太明显，需要检查程序。
- 3、使用的总内存数，以及最大指针，同上，在运行一段时间后，比如 1~2 小时，原则上应该稳定下来，不再增加，否则就可能有内存泄漏，需要检查程序。
- 4、下面的递归打印链表部分，所有的 all，即总块数，经过一定时间后，应该稳定，否则可能就是由泄漏。
- 5、这里的泄漏不仅仅是指内存申请了，忘了释放，也很可能是某一笔业务申请了内存，但业务本身被挂起，迟迟没有释放导致，因此，不能仅仅检查内存申请和释放的二元动作，还要检查一些长时间业务的最大时间是多少。
- 6、网络传输业务，一般都有超时，但如果几个小时，业务都被挂住，无法释放内存，则表明线程可能被挂死，很可能是重复加锁问题 (Double Lock)，需要检查所有对锁的使用。
- 7、如果锁检查无误，则很可能系统逻辑设计，某几笔业务出现了循环依赖关系，即 A 依赖 B，B 依赖 C，C 又依赖 A，这类属于逻辑错误，程序本身没有问题，因此，需要检查系统设计，检查所有逻辑依赖关系。
- 8、如果所有可能的原因都分析完，仍然找不到问题，则可能是业务量太大，并发的业务数过多所致，因此，需要检查目前测试的压力指标，如果太大，换更好的服务器测试，或者降低测试压力，修订程序能应对的最大压力指标。

7.3 内存指针注册管理模块

前面我们说过，一个成熟的内存池，仅仅做到内存块重用，减小内存碎片，是不够的。其更多的应该给程序员一个掌控所有内存块的手段，在出现内存泄漏的时候，能够自行报警，甚至能帮助程序员释放内存，才能尽量保证 100% 的安全。

因此，笔者的内存池，除了上述使用内存栈，真实地处理各种内存申请和释放请求，还有一个很重要的模块，即内存注册模块。

7.3.1 内存注册模块原理介绍

在笔者过去开发 C 和 C++ 工程程序时，很多时候，受制于指针的灵活性，出现了很多内存泄漏的 bug，这给笔者带来了很大的困扰，笔者长期思考的一个问题，就是如何一劳永逸地解决掉这个 bug 隐患。

这，甚至是笔者自行构建工程库的原始动机之一。

这个问题前后影响了笔者有三四年，直到有一天，笔者突发奇想，觉得可以利用注册和反注册机制来约束程序员的行为，这个问题才算有了解决之道。笔者是这么思考的：

- 1、内存泄漏是程序最大的 bug 隐患，必须予以消除。
- 2、内存泄漏最大的问题不是修改，只要找到泄漏点，修改很容易，最难的是查找泄漏原因。
- 3、查找内存泄漏，最困难的是，我们只看得到结果，看不到原因，即我们即使知道泄漏了多少内存，也很难找到是哪个模块申请的内存忘了释放。
- 4、如果我们能有个方法，记录下程序申请的每块内存，同时记录下其申请者，最后有内存泄漏时，拿着丢失的指针去找，就很容易找到是哪个模块忘了释放内存。
- 5、这个工作虽然很简单，但是不能让程序员来做，因为程序员是人，人是靠不住的，应该有一个程序模块，专门在运行期跟踪这件事情。
- 6、这个模块什么都不做，就是别人拿个指针，以及一段说明文字去注册，以后再申请释放掉即可，最后这个模块退出时，自动打印还没有释放的指针，以及其说明文字，就很容易地查找出，谁的指针忘了释放。

7.3.2 模块设计及类声明

基于上述原理，笔者开始设计这套注册和反注册机制，这个产品笔者做了很多代，从 2000 年左右开始，前后差不多写了近 40 版，中间交叉着 Windows 以及 Linux 等各种平台，VC 和 gcc 等各种语言的实现，但基本原理始终没有变过。

笔者这里给大家展示近期的一个版本，该版本本身就是跨平台通用的，各位读者可以放心使用。这里大家可以先看看核心数据结构

```
//说明文字的最大长度，该长度在多个点被调用，因此使用条件编译声明
#ifndef TONY_MEMORY_BLOCK_INFO_MAX_SIZE
#define TONY_MEMORY_BLOCK_INFO_MAX_SIZE 124
#endif
//内存注册类核心数据结构
typedef struct _TONY_XIAO_MEMORY_REGISTER_
{
    void* m_pPoint; //保留的指针
    char m_szInfo[TONY_MEMORY_BLOCK_INFO_MAX_SIZE]; //说明文字
}STonyXiaoMemoryRegister; //定义的变量类型
//习惯性写法，声明了结构体后，立即定义其尺寸常量
const ULONG STonyXiaoMemoryRegisterSize=sizeof(STonyXiaoMemoryRegister);
```

其实这个注册和反注册逻辑，是一个很典型的哈希型计算，即以指针为 key，查找其说明文字，由于内存指针，原则上在每次运行周期内，具有唯一性，因此，可以用来作为 key。

不过，笔者发现，很多低端的嵌入式设备，其编译器一般功能很弱，如果贸然使用语言提供的哈希计算工具，则不好做到跨平台通用。再加上这类查找功能一般很简单，不复杂，因此笔者一律自行实现。不借助其他工具。

这里特别要提一下管理逻辑：

1、一般这类应用，笔者习惯使用结构体数组来管理。

2、我们知道，一个程序的动态内存申请和释放，总是在不断变化的，但一般说来，总量不会变化太多，运行越久，越趋于一个平衡。因此，一个程序使用的指针，从并发总量来说，还是不会很大的，一个数组只要足够大，一般都能满足需求。

3、数组中，通常会有个额外变量 nUseMax，这个变量记录数组的使用情况，即已经多少单元被用到。

4、一个指针被 Add 进入，模块会首先检查内部数组 0~nUseMax 部分，看有没有被 Del 的空区，如果有，就利用空区存放，这样的好处是重用数组单元，避免溢出，且申请释放最频繁的指针，总是在数组开始部分，因此，遍历查找效率较高。

5、如果没有空区，则利用 nUseMax 指向的单元，存放数据，同时 nUseMax+1，这相当于插入到队列末尾。

6、Del 时，一般就是直接遍历整个数组，如果找到对应指针，将其单元清空，表示已经清除。

7、Modeify，表示在 ReMalloc 时，将旧指针替换成新指针，方便继续跟踪调用。

8、当最后模块退出时，析构函数会遍历整个数组，找出 0~nUseMax 之间，没有清空的指针，即应用模块忘了释放的指针，打印出来，提醒程序员检查 bug。

下面是类声明的实现：

```
//管理的最大指针个数，PC 平台一般建议 10000，但嵌入式平台，
//要根据具体情况分析，一般不建议超过 1024
#ifndef TONY_MEMORY_REGISTER_MAX
#define TONY_MEMORY_REGISTER_MAX 10000
#endif
//内存注册类
class CMemoryRegister
{
public:
    CMemoryRegister(CTonyLowDebug* pDebug);    //构造函数传输 debug 对象指针
    ~CMemoryRegister();
public:
    //添加一个指针及其说明文字
    void Add(void* pPoint,char* szInfo);
    //删除一个指针（该内存块被释放，失效）
    void Del(void* pPoint);
    //remalloc 的时候更新指针
    void Modeify(void* pOld,void* pNew);
    //打印信息函数，Debug 和观察性能用
    void PrintInfo(void);
private:
    //这是一个内部工具函数，把一段指针和说明文字，拷贝到上面定义的结构体中
    void RegisterCopy(STonyXiaoMemoryRegister* pDest,
        void* pPoint,char* szInfo);
```



```
private:
    CTonyLowDebug* m_pDebug;           //Debug 对象
    //请注意，由于注册类不管是注册，还是反注册动作，均包含写动作
    //此时，使用单写多读锁意义不大，因此使用普通锁来完成
    CMutexLock m_Lock;                 //线程安全锁
    //请注意这里，这里使用了静态数组，主要原因是考虑到内存注册类
    //是内存池的一个功能，不能再有动态内存申请，以免造成新的不稳定因素。
    //不过请注意，这个数组占用栈空间，对于嵌入式设备，可能没有这么大，要关注不要越界。
    STonyXiaoMemoryRegister m_RegisterArray[TONY_MEMORY_REGISTER_MAX];
    //这是数组使用标志，表示当前数组最大使用多少单元，
    //注意，这不是并发指针数，随着反注册的进行，数组中可能有空区
    int m_nUseMax;
    //另外一个统计，统计有史以来注册的最大指针，统计性能实用。
    void* m_pMaxPoint;
    //这是统计的当前在用指针数。
    int m_nPointCount;
};
```

7.3.3 构造函数和析构函数

构造函数主要负责初始化整个数组，相关变量。值得一提的是，在构造函数中，笔者使用了一个自定义宏 `TONY_CLEAN_CHAR_BUFFER(p)`，来清空一段字符串缓冲区。

在 C 和 C++ 语言中，通常情况下没有显式区分字符串缓冲区和二进制缓冲区，二者仅仅是内容理解不一样，从程序行为上是一致的。因此，C 和 C++ 语言并没有显式的清空缓冲区方法。因为 `'\0'` 也是合法的二进制数据。

但这给程序员带来很大困扰，一个缓冲区，如果不看程序上下文，很难判断它究竟是什么类型的缓冲区，这造成了 bug 的隐患。

笔者也是经过了长期的思考，才发现这个问题的重要性，因此，自行构建了一系列函数，将二进制行为和字符串行为尽量区别开，避免由于混用导致的 bug。

针对字符串缓冲区，笔者定义空串的构型，是第一个字节为 `'\0'`，这样方便后续程序的检查，因此，笔者定义了如下的字符串缓冲区清空宏定义。

```
//这里使用了一个宏 TONY_CLEAN_CHAR_BUFFER，其定义如下：
#define TONY_CLEAN_CHAR_BUFFER(p) (*(char*)p)='\0')
```

```

//构造函数
CMemoryRegister::CMemoryRegister(CTonyLowDebug* pDebug)
{
    m_pDebug=pDebug;
    //初始化变量
    m_pMaxPoint=null;
    m_nUseMax=0;
    m_nPointCount=0;
    //请注意这里，我们的结构体中，并没有单独设计标示使用与否的变量
    //因此，一般是以保存的指针是否为 null 来表示结构体是否被使用
    //这就要求，初始化时，先将所有的内存块指针均置为 null，方便后续函数判读
    int i=0;
    for(i=0;i<TONY_MEMORY_REGISTER_MAX;i++)
    {
        m_RegisterArray[i].m_pPoint=null;
        TONY_CLEAN_CHAR_BUFFER(m_RegisterArray[i].m_szInfo);
    }
}

```

析构函数在本模块中，非常重要，因为最关键的事后报警机制，即由析构函数实现。请各位读者关注其中对锁的使用

```

//析构函数
CMemoryRegister::~CMemoryRegister()
{
    int i=0;
    m_Lock.Lock(); //加锁
    {
        //打印统计的最大指针，提醒程序员
        TONY_DEBUG("CMemoryRegister: Max Register Point= 0x%p\n",m_pMaxPoint);
        //检索所有使用的结构体，如果有指针非 0，表示没有释放的，打印报警信息。
        //请注意，为了优化效率，这里循环的终点是 m_nUseMax，不检查没有使用的部分
        for(i=0;i<m_nUseMax;i++)
        {
            if(m_RegisterArray[i].m_pPoint)
            {
                //原则上，应该帮助应用程序释放，但考虑到，这个模块是基础支撑模块。
                //本析构函数执行时，进程即将关闭，因此，仅仅报警，不释放
                TONY_DEBUG("***** Memory Lost: [%p] - %s\n",
                    m_RegisterArray[i].m_pPoint,
                    m_RegisterArray[i].m_szInfo);
            }
        }
    }
    m_Lock.Unlock(); //解锁
}

```

这里顺便也介绍一下拷贝工具函数，该函数主要用于注册时，将传输的参数，保存到空闲的结构体中

```

void CMemoryRegister::RegisterCopy(
    STonyXiaoMemoryRegister* pDest,    //目标结构体指针
    void* pPoint,                      //待拷贝的注册指针
    char* szInfo)                      //待拷贝的说明文字
{
    pDest->m_pPoint=pPoint;
    if(szInfo)                        //由于 szInfo 可以是 null, 因此需要加判断
    {
        //请注意这里对 SafeStrcpy 的使用, 拷贝永远是安全的, 程序减少很多判断, 显得很简洁
        SafeStrcpy(pDest->m_szInfo,szInfo,TONY_MEMORY_BLOCK_INFO_MAX_SIZE);
    }
    else
        TONY_CLEAN_CHAR_BUFFER(szInfo); //如果为空, 则缓冲区置为空字符串
}

```

如上所示, 由于析构函数中设计了报警机制, 加入我们一个模块函数 CClass::DoIt 中, 申请了内存, 并以其函数名做了注册说明, 一旦忘了释放, 则在程序最后退出时, 会自动报警:

```
***** Memory Lost: [0XXXXXXXXX] - CClass::DoIt()
```

程序员立即就可以看到, 因此, 可以迅速寻找到 bug 点 CClass::DoIt 检查, 并及时修正 bug。

这个功能看似简单, 但非常有效, 在过去的商用工程开发中, 无数次帮助笔者以及团队伙伴, 避免了内存泄漏的 bug, 极大地保证了产品质量。

提示: 这里也可以提示各位读者一个思路, 很多看似复杂的问题, 比如内存泄漏, 很难通过常规的方法解决, 程序员完全可以根据自己的需求, 设计一些模块, 功能和方法, 来帮助自己直观地解决问题, 这类方法一般并不难, 仅仅是多想一点即可, 请各位读者深思。

7.3.4 Add 函数

Add 即本类的注册动作, 是最重要的功能函数之一。

```

//注册方法
void CMemoryRegister::Add(void* pPoint,char* szInfo)
{
    int i=0;
    m_Lock.Lock(); //加锁
    {
        //完成统计功能，如果新注册的指针比最大指针大，更新最大指针
        if(pPoint>m_pMaxPoint) m_pMaxPoint=pPoint;
        //循环遍历已经使用的数组区域，寻找因 Del 导致的空区，重复使用
        //注意，循环的终点是 m_nUseMax，而不是数组最大单元
        for(i=0;i<m_nUseMax;i++)
        {
            //请注意，如果结构体保存的指针为空，判定未使用
            if(!m_RegisterArray[i].m_pPoint)
            {
                //统计在用指针总数
                m_nPointCount++;
                //调用拷贝功能函数，执行真实的拷贝动作，保存传入的信息，即注册
                RegisterCopy(&(m_RegisterArray[i]),pPoint,szInfo);
                //功能完成，直接跳到最后，返回
                goto CMemoryRegister_Add_End_Process;
            }
        }
        //这是第二逻辑段落，即使用区域无空区，需要增补到最尾
        //首先，检查是否数组越界，如果越界，报警
        if(TONY_MEMORY_REGISTER_MAX<=m_nUseMax)
        {
            //使用 Debug 模块报警，原则上，程序员如果在 Debug 信息中看到这一句
            //表示该程序并发的指针数超过了数组限制，可以考虑
            //增加 TONY_MEMORY_REGISTER_MAX 的值。
            TONY_DEBUG("***ERROR*** CMemoryRegister is full!\n");
            goto CMemoryRegister_Add_End_Process;
        }
        //这是讲注册内容拷贝到队列最尾的逻辑
        RegisterCopy(&(m_RegisterArray[m_nUseMax]),pPoint,szInfo);
        m_nPointCount++;
        m_nUseMax++;
    }
CMemoryRegister_Add_End_Process:
    m_Lock.Unlock(); //解锁
}

```

7.3.5 Del 函数

Del 函数反注册一个指针，任何指针，经过反注册，即脱离了注册类的管理，不再提供报警服务，一般情况下，Del 方法会和 Free 联用，但也有特例，这在下文有详细论述。

```

//反注册函数
void CMemoryRegister::Del(void* pPoint)
{
    int i=0;
    m_Lock.Lock(); //加锁
    {
        //寻找在用内存区域
        for(i=0;i<m_nUseMax;i++)
        {
            以内部保存的内存指针是否相等，判断是否是对应结构体
            if(pPoint==m_RegisterArray[i].m_pPoint)
            {
                //统计值-1
                m_nPointCount--;
                //清空逻辑，本结构体也就空闲出来，等待下次 Add 重用
                m_RegisterArray[i].m_pPoint=null;
                TONY_CLEAN_CHAR_BUFFER(m_RegisterArray[i].m_szInfo);
                //跳出循环，返回
                goto CMemoryRegister_Del_End_Process;
            }
        }
    }
    CMemoryRegister_Del_End_Process:
    m_Lock.Unlock(); //解锁
}

```

7.3.6 Modeify 函数

Modeify 函数在笔者的原始设计中并不存在，不过在笔者设计内存栈的 ReMalloc 功能时，发现一个新的需求，即一个函数申请的指针，中途可能会变化，这时候需要更新内存注册类里的数据，否则，可能产生误报警。因此，特地添加了这个方法。

```

//修改注册指针的值
void CMemoryRegister::Modeify(void* pOld,void* pNew)
{
    int i=0;
    m_Lock.Lock();
    {
        //由于要切换指针，因此需要做一次统计
        if(pOld>m_pMaxPoint) m_pMaxPoint=pOld;
        for(i=0;i<m_nUseMax;i++)
        {
            //注意，所有的检索以内存块指针作为依据，相当于哈希上的 key
            if(pOld==m_RegisterArray[i].m_pPoint)
            {
                //仅仅修订指针，不修订说明文字
                m_RegisterArray[i].m_pPoint=pNew;
                goto CMemoryRegister_Del_End_Process;
            }
        }
    }
    //这是一个显式的报警，如果一个指针没有找到，而应用层又来请求修改
    //这表示应用程序有 bug，因此，debug 提醒程序员修改 bug
    TONY_DEBUG("***ERROR*** CMemoryRegister::Modeify(): I can\'t found
point!\n");
CMemoryRegister_Del_End_Process:
    m_Lock.Unlock();
}

```

7.3.7 PrintInfo 函数

内存注册类的 PrintInfo 相对比较简单，仅仅输出本类几个关键管理变量即可。

```

//信息打印函数
void CMemoryRegister::PrintInfo(void)
{
    m_Lock.Lock();
    {
        TONY_XIAO_PRINTF("memory pool: %d / %d, biggest=%p\n",
            m_nPointCount,           //在用指针总数
            m_nUseMax+1,             //注册数组使用范围
            m_pMaxPoint);            //统计的最大指针
    }
    m_Lock.Unlock();
}

```

7.3.8 内存注册模块的深入使用

各位读者看到本小节内容，可能会发现部分功能设置和内存栈有重合之处，最明显的就是 pMaxPoint 这个设计，即统计当前使用的最大指针。看起来和内存栈的设计完全一致，显得没有必要。

其实这是有必要的，笔者之所以没有把这个内存注册模块和内存栈设计到一起，而是分割为两个类，有一个最核心的目的前面没有说清楚。

我们知道，C 和 C++ 的程序，主要有动态内存申请，一种是基于纯内存块的 `malloc` 和 `free`，而另外一类是 C++ 的动态对象创建和摧毁。

我们前面已经说过，内存池不具备自动调用构造函数和析构函数功能，因此，无法替代 C++ 完成对象的创建和摧毁，但我们也同意，很多时候，无论我们如何优化设计，在 C++ 代码中，对象的动态创建是不可避免的。

这就造成了，内存池虽然很努力地在管理所有的内存块，但另外一大类，即动态对象，仍然没有办法管理，也就无法自动为程序员提出报警，帮助程序员 `Debug`。

因此，笔者经过思考，决定把内存块的管理对象和内存指针的注册使用对象分开，前者，内存栈，主要针对纯正的二进制内存块进行管理，而内存指针注册对象，则针对所有的内存块，以及动态对象指针进行管理，帮助程序员管理 C++ 的对象指针。

这样，一方面，我们的程序申请内存块，可以使用这个内存注册模块管理，另一方面，即使其他的应用模块，动态 `new` 出一个对象，也可以将这个对象指针注册进来，实施管理。当对象忘了摧毁时，本模块一样会提出告警信息。

这样，我们回过头看，可以发现，内存注册模块中统计的最大指针，和内存栈中的最大指针，不一样，本模块的最大指针，很可能是一个对象指针，而不是内存块的指针，也就很可能比内存栈统计数据要大。

在下面的章节，我们可以看到内存注册模块如何协助管理对象指针。

7.4 Socket 注册管理模块

由于有了内存指针注册管理模块，笔者发现内存泄漏的管理变得很容易，因此，就形成了思维套路，遇到容易遗忘的二元性动作，总是喜欢以上述思路去做注册管理，避免类似的 `bug`。

在商用数据传输工程中，笔者发现 `socket` 的管理也是个让人很头大的问题。商用工程，强调业务逻辑层次化，使用协议栈来管理传输。业务层和功能层分得很清晰。比如，通常情况下，`Listen` 线程，监听到一个 `socket` 连接请求，并且 `accept` 出一个新的 `socket`，并不会自己来处理该 `socket` 的传输行为，而是将其交给专门的业务处理线程来处理，自己则专心地应对下一个连接请求。

这说明，在商用数据传输工程中，`socket` 的管理，通常都不会遵守“谁申请，谁释放”原则，一个 `socket`，很可能由 `Listen` 线程产生，然后交给请求响应线程接收请求，再将其连同请求数据一起放入任务队列，由任务处理机从队列中检索，执行请求，最后，将回应利用该 `socket` 传回给客户端，然后关闭 `socket`，宣告本笔业务交易结束。

这中间 `socket` 经过了几次倒手，分别由几个线程，异步地处理其业务，在程序代码上，这也分属不同的函数体，中间很容易遗忘了关闭。

再者，前文我们说过，在进行业务管理时，我们经常会面临一笔业务，多次 `socket` 连接的复杂业务逻辑，因此，我们一般使用 `Session`（会话）这个概念来管理一笔业务交易。也就是说，业务交易通常是与业务绑定的，与 `socket` 并无直接的伴生关系，二者生命周期也不一致，因此很难利用业务模型来管理 `socket` 的生死，即我们不太可能通过一个业务类的构造函数，或者析构函数来自动创建和关闭 `socket`。

但另外一方面，我们知道，根据 TCP/IP 协议的规约，一个设备的 `socket` 理论数值只有 65536 个，通常根据机器的内存大小，甚至还低于这个数字，如果 `socket` 发生泄漏，即忘了关闭，机器很容易因为 `socket` 资源被耗尽，最终导致无法响应服务，形同挂死。

以上所有问题，说明我们有必要把 socket 也做个注册管理，实现 socket 的自动报警，避免出现 socket 的泄漏。因此，笔者放在上文的内存指针注册管理，构造了一个 socket 的注册管理模块。

7.4.1 类声明

本类是 Linux 和 Windows 通用，内部使用了很多跨平台的定义，如 Linux_Win_SOCKET。

```
//核心管理数据结构
typedef struct _TONY_XIAO_SOCKET_REGISTER_
{
    Linux_Win_SOCKET m_nSocket;           //保存的 socket
    char m_szInfo[TONY_MEMORY_BLOCK_INFO_MAX_SIZE]; //说明文字，长度同上
} STonyXiaoMemoryRegister;
//结构体尺寸常量
const ULONG STonyXiaoSocketRegisterSize=sizeof(STonyXiaoSocketRegister);
//socket 注册管理类
class CSocketRegister
{
public:
    CSocketRegister(CTonyLowDebug* pDebug); //构造函数，传入 debug 对象指针
    ~CSocketRegister();
public:
    void PrintInfo(void); //信息打印函数
public:
    //注册一个 socket，以及其说明文字
    void Add(Linux_Win_SOCKET s,char* szInfo=null);
    //反注册一个 socket，如果内部没找到，返回 false
    bool Del(Linux_Win_SOCKET s);
private:
    CTonyLowDebug* m_pDebug; //debug 对象指针
    CMutexLock m_Lock; //锁，这里同样使用普通锁
    //保存注册信息的结构体数组，大小也同内存指针注册类
    STonyXiaoSocketRegister m_RegisterArray[TONY_MEMORY_REGISTER_MAX];
    //管理变量
    int m_nUseMax;
    //统计变量
    Linux_Win_SOCKET m_nMaxSocket; //注册过的最大 socket，
    int m_nSocketUseCount; //在用的 socket 总数
};
```


7.4.2 构造函数和析构函数

//构造函数

```
CSocketRegister::CSocketRegister(CTonyLowDebug* pDebug)
{
    m_pDebug=pDebug;                                //保存 debug 对象指针
    //初始化各种变量
    m_nMaxSocket=Linux_Win_InvalidSocket;
    m_nSocketUseCount=0;
    m_nUseMax=0;
    int i=0;
    for(i=0;i<TONY_MEMORY_REGISTER_MAX;i++)
    {
        //同上，我们以一个结构体中保留的 socket= Linux_Win_InvalidSocket
        //作为结构体未使用的标志
        m_RegisterArray[i].m_nSocket=Linux_Win_InvalidSocket;
        TONY_CLEAN_CHAR_BUFFER(m_RegisterArray[i].m_szInfo);
    }
}
```

//析构函数

```
CSocketRegister::~CSocketRegister()
{
    int i=0;
    m_Lock.Lock();
    {
        TONY_DEBUG("CSocketRegister: Max Socket Count=%d, Max Socket=%d\n",
            m_nUseMax,m_nMaxSocket);                //打印关键信息
        for(i=0;i<m_nUseMax;i++)
        {
            if(SocketIsOK(m_RegisterArray[i].m_nSocket))
            {
                //退出时，如果发现有用 socket，报警，并代为释放
                TONY_DEBUG("***** Socket Lost: [%d] - %s\n",
                    m_RegisterArray[i].m_nSocket,
                    m_RegisterArray[i].m_szInfo);
                //这是释放 Socket 的语句
                _Linux_Win_CloseSocket(m_RegisterArray[i].m_nSocket);
            }
        }
    }
    m_Lock.Unlock();
}
```

```

//这里给大家提示一点跨平台的关闭 Socket 方法
//请注意，Windows 下和 Linux 下，错误的 socket 表示值不一样，需要条件编译分别确定
#ifdef WIN32
    #define Linux_Win_InvalidSocket INVALID_SOCKET
#else // not WIN32
    #define Linux_Win_InvalidSocket -1
#endif
//判断 socket 是否合法的通用函数，这个函数设置的目的是收拢所有的判断，方便以后修改
bool SocketIsOK(Linux_Win_SOCKET nSocket)
{
    //某些系统的 socket 判断，是以 0~65535 的绝对值判断，这里实现，暂时不用
    // if(0>nSocket) return false;
    // if(65535<nSocket) return false;
    //原则上，只要不等于 Linux_Win_InvalidSocket 的其他整数，均可视为合法 socket
    if(Linux_Win_InvalidSocket==nSocket) return false;
    return true;
}
//真实地关闭一个 socket
void _Linux_Win_CloseSocket(Linux_Win_SOCKET nSocket)
{
    if(!SocketIsOK(nSocket)) return;
#ifdef WIN32
    closesocket(nSocket);
#else // not WIN32
    close(nSocket);
#endif
}

```

7.4.3 Add 函数

Add 函数和内存指针管理模块的 Add 有较大差别，请各位读者千万不要弄混。

我们知道，Socket 在其生命周期内，通常有多个所有者，即多个线程函数会利用该 Socket 做事情，因此，Socket 的注册方法原则上应该允许应用层修改说明文字，以便及时跟踪当前是哪个函数在处理 Socket。

这就说明 Add 在本模块中，是个复合功能的函数，遇到一个注册请求，首先会检测该 Socket 是否已经存在于本地，如果是，则修改其说明文字，否则视为添加请求，添加前，也是先寻找空区，如果有，则使用该空区注册，没有，则添加到队列最后。

```

//Socket 注册管理模块的注册函数
void CSocketRegister::Add(Linux_Win_SOCKET s,char* szInfo)
{
    int i=0;
    m_Lock.Lock(); //加锁
    {
        //统计行为, 统计最大的 socket 值供查阅
        if(!SocketIsOK(m_nMaxSocket)) m_nMaxSocket=s;
        else if(s>m_nMaxSocket) m_nMaxSocket=s;
        //先试图修改, 遍历使用区域
        for(i=0;i<m_nUseMax;i++)
        {
            if(m_RegisterArray[i].m_nSocket==s)
            {
                if(szInfo) //拷贝说明文字
                    SafeStrcpy(m_RegisterArray[i].m_szInfo,
                                szInfo,TONY_MEMORY_BLOCK_INFO_MAX_SIZE);
                //注意, 修改不是添加, 因此, 这里的 socket 使用统计不增加
                //m_nSocketUseCount++;
                goto CSocketRegister_Add_End_Process;
            }
        }
        //再试图插入
        for(i=0;i<m_nUseMax;i++)
        {
            if(!SocketIsOK(m_RegisterArray[i].m_nSocket))
            {
                m_RegisterArray[i].m_nSocket=s;
                if(szInfo) //拷贝说明文字
                    SafeStrcpy(m_RegisterArray[i].m_szInfo,
                                szInfo,TONY_MEMORY_BLOCK_INFO_MAX_SIZE);
                //这是实实在在的添加到空区, 因此, 统计变量+1
                m_nSocketUseCount++;
                goto CSocketRegister_Add_End_Process;
            }
        }
        //最后无空区可以使用呢, 试图追加到最后
        if(TONY_MEMORY_REGISTER_MAX>m_nUseMax)
        {
            m_RegisterArray[m_nUseMax].m_nSocket=s;
            if(szInfo) //拷贝说明文字
                SafeStrcpy(m_RegisterArray[m_nUseMax].m_szInfo,
                            szInfo,TONY_MEMORY_BLOCK_INFO_MAX_SIZE);
            //使用区域指针+1
            m_nUseMax++;
            //统计变量+1
            m_nSocketUseCount++;
        }
        //注册区满了, 报警, 并发的 socket 数量超限, 程序员有必要扩大缓冲区
        else TONY_DEBUG("CSocketRegister::Add(): Pool is full!\n");
    }
    CSocketRegister_Add_End_Process:
    m_Lock.Unlock(); //解锁
}

```

7.4.4 Del 函数

```
//反注册函数，即将一个 socket 的注册信息，从内部管理数据区删除
bool CSocketRegister::Del(Linux_Win_SOCKET s)
{
    bool bRet=false;
    int i=0;
    m_Lock.Lock();           //加锁
    {
        for(i=0;i<m_nUseMax;i++)
        {
            //遍历使用区，检索 socket
            if(m_RegisterArray[i].m_nSocket==s)
            {
                //注意清空动作，把 socket 置为 Linux_Win_InvalidSocket
                //这项，下次 Add 可以重复使用该空区
                m_RegisterArray[i].m_nSocket=Linux_Win_InvalidSocket;
                TONY_CLEAN_CHAR_BUFFER(m_RegisterArray[i].m_szInfo);
                //修订统计变量，并发 socket 数量-1
                m_nSocketUseCount--;
                bRet=true;
                goto CSocketRegister_Del_End_Process;
            }
        }
    }
    CSocketRegister_Del_End_Process:
    m_Lock.Unlock();         //解锁
    return bRet;
}
```

7.4.5 PrintInfo 函数

```
//内部信息打印函数
void CSocketRegister::PrintInfo(void)
{
    m_Lock.Lock();           //加锁
    {
        TONY_XIAO_PRINTF("socket pool: %d / %d, biggest=%d\n",
            m_nSocketUseCount,    //并发的 socket 数量统计
            m_nUseMax+1,          //内存结构体数组使用量标示
            m_nMaxSocket);        //统计到的最大 socket
    }
    m_Lock.Unlock();         //解锁
}
```

7.5 内存池类

本章的前面章节，我们详细论述了实现一个内存池各个功能模块的细节，当然，上述模块功能还比较零散，要想真正整合到一起，形成一个完整意义上的内存池，还需要做一些聚合工作，即本小节介绍的内存池类。

在本章开始的时候，我们介绍了内存池类的基本需求，帮助程序员检查内存泄漏的 bug，同时，实现内存块重用，避免内存碎片。

在前文，我们主要针对各个功能的实现，做出了具体的工具类和函数，而内存池的类设计，更多的是 C++ 的聚合逻辑，即将多个子类，聚合到一个父类中，共同完成功能。

7.5.1 类声明

```
//内存池类
class CTonyMemoryPoolWithLock
{
public:
    //构造函数
    //考虑到某些成熟的代码模块，已经经过测试，确定没有 bug，可以考虑在这里
    //关闭注册功能，这样，减少了每次内存申请和释放时，注册和反注册的逻辑，可以提升效率
    CTonyMemoryPoolWithLock(CTonyLowDebug* pDebug, //传入的 Debug 对象指针
        bool bOpenRegisterFlag=true);           //不开启注册功能的标志
    ~CTonyMemoryPoolWithLock();
    ////////////////////////////////////////
    //指针管理
public:
    //注册指针，实现管理，注意需要说明文字
    void Register(void* pPoint,char* szInfo);
    void UnRegister(void* pPoint);
private:
    CMemoryRegister* m_pRegister;           //指针注册管理对象
    ////////////////////////////////////////
    //Socket 管理
public:
    //注册 Socket，实现管理，注意需要说明文字
    void RegisterSocket(Linux_Win_SOCKET s,char* szInfo=null);
    //代应用程序执行 Close Socket，所有 Socket 的 Close 由此处完成
    void CloseSocket(Linux_Win_SOCKET& s);    //关闭 Socket
private:
    CSocketRegister* m_pSocketRegister;      //Socket 注册管理对象
    ////////////////////////////////////////
    //公共管理
```

```

public:
    //作为一项优化, 如果 app 设置这个关闭标志,
    //所有的 free 都直接 free, 不再保留到 stack 中。
    //加快退出速度。
    void SetCloseFlag(bool bFlag=true);
    //重新分配一个指针的空间, 默认拷贝原始数据到新空间。
    //考虑到大家可能会使用 p=pMemPool->ReMalloc(p,1024);这种形式, 为了防止内存泄露
    //如果 pPoint 没有被找到, 或者新指针分配失败, 老指针会被 free
    void* ReMalloc(void* pPoint,int nNewSize,bool bCopyOldDataFlag=true);
    //分配一个块, 注意, 需要一段说明文字, 说明指针用途, 方便注册跟踪
    void* Malloc(int nSize,char* szInfo=null);
    //释放一个块
    void Free(PVOID pBlock);
    //显示整棵内存树的内容
    void PrintTree(void);
    //关键信息显示
    void PrintInfo(void);
    CTonyMemoryStack* m_pMemPool;                //内存栈对象
    CTonyLowDebug* m_pDebug;
};

```

7.5.2 构造函数和析构函数

细心的读者, 可能在本类的构造函数中, 发现了一些不太符合 C 和 C++ 无错化程序设计方法的设计, 如每个子类对象指针的 new 行为, 均没有做成功判断, 而是直接使用。

笔者在此设计时, 主要考虑到, 内存池通常是一个应用服务程序最底层的设计, 它总是最先实例化开始运行, 并且最后一个析构, 其生命周期, 仅次于 Debug 对象。

这说明, 内存池的构造函数开始运行时, 应用程序的主逻辑尚未开始执行, 大规模的动态内存申请, 也还没有开始, 此时, 操作系统分配给本进程的 2G 地址空间, 几乎为空, 本类中的 new 行为, 总是能运行成功, 因此无需做详细的校验。

从另外一个角度说, 内存池是整个软件系统的运行核心, 它都无法正确实例化, 则系统可能已经出了很严重的问题, 程序根本无法运行。即使校验出失败, 也没有什么事情好做, 程序只有崩溃了事。等待程序员做进一步的系统设计分析。

另外, 大家可能已经注意到, 内存池中并没有设计锁, 这是因为各个子对象均内部实现了锁保护, 并且, 锁的原则都不一致, 内存池类是一个纯粹的组织类, 内部并不实现任何具体的业务, 仅仅是转发请求, 因此, 无需再加锁, 以免影响效率。

```

//构造函数
CTonyMemoryPoolWithLock::CTonyMemoryPoolWithLock(
    CTonyLowDebug* pDebug,
    bool bOpenRegisterFlag)
{
    m_pDebug=pDebug; //保存 debug 对象指针
    m_pMemPool=new CTonyMemoryStack(m_pDebug); //实例化内存栈对象
    m_pRegister=null; //初始化各个指针变量
    m_pSocketRegister=null;
    m_pLockRegister=null;
    if(bOpenRegisterFlag)
    { //根据标志位, 决定是否实例化各个注册对象
        m_pRegister=new CMemoryRegister(m_pDebug);
        m_pSocketRegister=new CSocketRegister(m_pDebug);
    }
    //打印内存池正确启动标志
    TONY_DEBUG("Tony.Xiao. Memory Pool Open, register flag=%d\n",
        bOpenRegisterFlag);
}
//析构函数
CTonyMemoryPoolWithLock::~~CTonyMemoryPoolWithLock()
{
    //依次摧毁各个子对象
    if(m_pRegister) //请读者注意删除对象的标准写法
    {
        delete m_pRegister;
        m_pRegister=null;
    }
    if(m_pSocketRegister)
    {
        delete m_pSocketRegister;
        m_pSocketRegister=null;
    }
    if(m_pMemPool)
    {
        delete m_pMemPool;
        m_pMemPool=null;
    }
    //打印内存池正确结束标志
    TONY_DEBUG("Tony.Xiao. Memory Pool Close.\n");
}

```

7.5.3 内存栈公有方法

如前文所述, 内存栈的公有方法, 主要包括 **Malloc**, **Free**, **ReMalloc** 等几个方法, 内存池主要做方法转移。

不过, 值得一提的是, 内存池中, 为成功申请的每一根指针, 自动做了注册和反注册管理, 方便应用程序使用。

```
//设置退出标志，加速内存栈的释放过程
void CTonyMemoryPoolWithLock::SetCloseFlag(bool bFlag)
{
    if(m_pMemPool) m_pMemPool->SetCloseFlag(bFlag);
}
```

```
//Malloc 函数
void* CTonyMemoryPoolWithLock::Malloc(int nSize,char* szInfo)
{
    void* pRet=null;
    if(m_pMemPool)
    {
        pRet=m_pMemPool->Malloc(nSize);    //调用内存栈实现内存分配
        if(pRet) Register(pRet,szInfo);    //如果指针有效，自动注册
    }
    return pRet;
}
```

```
//Free 函数
void CTonyMemoryPoolWithLock::Free(PVOID pBlock)
{
    if(m_pMemPool)
        m_pMemPool->Free(pBlock);    //调用内存栈实现 Free
    UnRegister(pBlock);    //反注册指针
}
```

```
//ReMalloc 函数
void* CTonyMemoryPoolWithLock::ReMalloc(
    void* pPoint,
    int nNewSize,
    bool bCopyOldDataFlag)
{
    void* pRet=null;
    if(m_pMemPool)
    {    //调用内存栈重定义内存区域大小
        pRet=m_pMemPool->ReMalloc(pPoint,nNewSize,bCopyOldDataFlag);
        if(m_pRegister)
        {
            if(pRet)    //如果重分配成功，在注册对象中 Modeify 新的指针
                m_pRegister->Modeify(pPoint,pRet);
            else    //如果失败，由于老指针已经摧毁，因此需要从注册对象中反注册旧指针
                m_pRegister->Del(pPoint);
        }
    }
    return pRet;
}
```


7.5.4 指针管理方法

前文中我们说了，内存池的指针注册管理机制，不仅仅要应对内部的动态内存分配需求，还要提供公有接口和方法，帮助应用层实现动态创建的对象指针的注册管理，避免对象指针的泄漏。

因此，本类中一方面在 **Malloc** 和 **Free** 等内存块分配释放接口函数做了指针注册管理，还单独设计了公有方法接口，供应用层使用。细心的读者可能注意到，**Malloc** 和 **Free** 中的注册管理，其实是调用这两个方法完成的。

```
//指针注册方法
void CTonyMemoryPoolWithLock::Register(void* pPoint,char* szInfo)
{
    if(m_pRegister)
    {
        m_pRegister->Add(pPoint,szInfo);
    }
}
//指针反注册方法
void CTonyMemoryPoolWithLock::UnRegister(void* pPoint)
{
    if(m_pRegister)
    {
        m_pRegister->Del(pPoint);
    }
}
```

7.5.5 Socket 管理方法

同上，内存池也提供了对 **Socket** 资源的注册管理。不过，由于 **socket** 的特殊性，本类没有提供反注册方法，而是提供了一个公有的 **socket** 关闭方法，将全局所有 **socket** 关闭动作收拢到这里，实现统一的反注册+关闭机制。

```
//应用程序注册一个 socket 开始管理，注意，一个 socket，可能被注册多次，以修改其说明文字
//本函数是可以多次重入的。
void CTonyMemoryPoolWithLock::RegisterSocket(Linux_Win_SOCKET s,char* szInfo)
{
    if(m_pSocketRegister)
    {
        m_pSocketRegister->Add(s,szInfo);
    }
}
```

```

//公有的关闭 Socket 方法，反注册+关闭。
void CTonyMemoryPoolWithLock::CloseSocket(Linux_Win_SOCKET& s)
{
    if(SocketIsOK(s))
    {
        if(m_pSocketRegister)
        {
            if(!m_pSocketRegister->Del(s))
            {
                //这是一个很重要的报警，如果一个 socket，经检查，没有在内部注册
                //但是应用程序却 call 这个方法，就表示程序内存在没有注册的 socket 情况
                //说明 socket 的管理不完整，有遗漏，程序员有必要检查程序
                //还有一种可能，这个 socket 被两个程序关闭，第一次关闭时，
                //其信息已经反注册，第二次就存在找不到的现象
                //这说明程序员对 socket 的关闭有冗余行为，虽然不会有什么问题，
                //但建议最好修改。
                TONY_DEBUG("CTonyMemoryPoolWithLock::CloseSocket(): \
                    Socket %d is not registered! but I have close it yet!\n",
                    s);
            }
        }
        //真实地关闭 socket
        _Linux_Win_CloseSocket(s);
        s=Linux_Win_InvalidSocket;    //关闭完后，立即赋值
    }
}

```

7.5.6 PrintfInfo 方法

内存池的 PrintfInfo 方法，主要调用各个子类的打印函数，打印相关信息。

```

//调用内存栈功能，打印详细跟踪信息
void CTonyMemoryPoolWithLock::PrintTree(void)
{
    if(m_pMemPool) m_pMemPool->PrintStack();
}
//打印各个子对象信息
void CTonyMemoryPoolWithLock::PrintInfo(void)
{
    if(m_pSocketRegister) //打印 Socket 注册对象信息
        m_pSocketRegister->PrintInfo();
    if(m_pRegister) //打印指针注册对象信息
        m_pRegister->PrintInfo();
    if(m_pMemPool) //打印内存栈对象信息
        m_pMemPool->PrintInfo();
}

```

7.6 内存管理的深层次含义

经过本章的讲解，大家应该对使用内存池的原因，内存池实现的基本原理，以及如何实现一个内存池相关的知识，已经基本了解了。

从前面的代码中，大家可以发现，内存池作为一个概念，往往在书中看到时，我们会觉得很神秘，也很费解，但真正实现出来，大家又可以发现，其实还是很简单的。

这是正常现象，很多程序设计领域，非常重要，非常神秘的设计，仔细分析下来，其成因和实现机制并不复杂，关键是程序员的理念是否到位。这里，笔者认为有必要给各位读者再次强调内存池体现出来的理念。

7.6.1 资源重用的理念

首先，内存池体现的是资源重用理念，我们知道，没有哪个计算机系统，敢说自己的资源无限，我们做商用系统，系统分析的第一步，往往就是数据边界确定，即从程序设计的一开始，我们就有必要保证处理数据不越界。这是系统保证稳定的基础。

而我们知道，商用工程中，很多时候面临的是无穷无尽的动态请求，如何以有限的资源应对近乎无限的客户需求，资源重用势在必行。即程序设计的原则，当一笔交易发生，应该尽快完成，成功，给出正确的回应，失败，返回一个错误，但不管成功与否，都应该尽快结束交易，释放资源，以应对下一次请求。

同时，释放的资源，不要频繁和系统交互，更多地是内部实现资源池，临时保存，供下一次申请使用。以此来避免内存碎片等系统不足。

7.6.2 注册和反注册机制

注册和反注册机制，可以说是笔者自己创造出来的，看起来，这给程序员带来了麻烦，每次申请内存，需要说明申请者身份，一般就是函数名的文本表示，笔者建议，包括动态对象的建立，其指针也应该使用内存池注册后使用，以监督泄漏行为。

但大家可以想象到，仅仅这一点不方便之处，带来的却是永无内存泄漏的好处，笔者认为，这一点不便，相对其获得好处来说，是值得的。

正是因为这套机制的存在，笔者才在近 10 年的时间里，从没有出现过内存泄漏，即使某些时候因为笔误或遗忘犯了错误，程序也会在第一次运行结束时，自动报警，并提示泄漏的模块名，笔者可以第一时间改正 bug。

这说明，很多功能强大的设计，其实实现起来非常简单。仅仅是程序员想不想得到。笔者写作本书，并不仅仅希望各位读者能把代码拿来就用，还希望帮助各位读者，找到解决问题的思路，下次遇到类似问题，可以自行创造出方法来解决。

这一点请各位读者关注。

7.6.3 静态资源的管理思路

内存池，经过笔者的不断扩充，尤其是加入了对 Socket 的管理之后，更像一个静态资源池。对内存池的维护，笔者可能会不断地进行下去，以后的程序设计生涯中，如果遇到类似的针对有限资源防止泄露的问题，笔者大约都会整合到内存池中，加以解决。

例如，笔者近期就在探索如何将锁的行为，也用内存池的注册和反注册机制加以管理，防止出现 DoubleLock，不过这个功能尚未完成，笔者这里不再赘述，有兴趣的读者，可以自行研究实现。

7.7 被动池的常见组织形式

内存池，也可以视为前文所述被动池的典型样例，大家从中可以看到一个被动池的真实实现过程。

很多时候，我们在讨论池技术时，看似很神秘，但从本章大家可以看到，池的实现有时候非常简单，根本无需使用什么复杂的算法。

在注册和反注册机制前几版中，笔者曾经尝试过很多的复杂方法，比如哈希算法，比如平衡二叉树，最后改来改去，却发现最简单的就是最有效的，简简单单一个静态数组，几个遍历循环，就把功能实现了，而且效率还不低。

最关键的是，简单的数据结构，避免了大量的动态内存申请，这极大地提高了内存池自身的安全性，程序稳定性得到了保障。

7.7.1 被动池的数据特性及需求分析

这里，笔者认为有必要为大家介绍一下被动池的常见组织形式。

通常情况下，池被用来描述一堆资源的集合，特别是被动池尤其如此。其最核心的特点，就是池内资源的完全相似性，即属于同一个池内的所有资源，具有完全相同的服务特性，一个应用请求，请求池内任何一个资源，所获得的服务效果和服务品质，是完全一样的。

因此，如果我们要以池技术来管理资源，首先要做到的就是池内成员的完全相似性。这在 C 和 C++ 中，完全可以使用同一个数据描述结构体，或者同一个类的多个实例化对象来实现。

在本章中，大家看到的内存池这个被动池，其核心，就是内存块的描述结构体。

正是因为池的这个特性，因此在池内对象的筛选算法上，首先就是允许无序筛选，即允许随机筛选，这给被动池的算法和数据结构组织，带来很大便利。

我们知道，通常我们遇到的数据传输请求，都有“先进先出”的特性，即数据处理的顺序要遵守数据产生的顺序，这在算法上，通常需要用队列来完成，我们知道，队列的计算稍稍复杂，会给程序的性能带来一点影响。

但被动池通常没有这个限制，对于资源的选择，“先进先出”和“后进先出”都可以，因此，在本章中，大家可以看到，笔者使用了“栈”来管理资源。

在以后的工作中，各位读者可能会碰到其他的池开发需求，但只要是被动池，一般来说，都可以借鉴本章的方法来解决。这个解决方案与操作系统平台无关，与语言无关，可以认为是一种通用的解决方案。

7.7.2 动态与静态被动池的差异性

在本章中，其实笔者给大家展示了两种被动池，一种是内存栈的动态被动池实现，其管理的资源是动态申请的，并有能力动态释放的（仅仅是内存池的管理要求，使其不被动态释放）。

同时，大家可能注意到，我们关于内存指针和 Socket 的注册管理模块，也是被动池的一种表现形式，它对外提供的服务是指针（或 Socket），以及其说明文字的可存储，可修改能力。这直接体现在由结构体组成的数组，STonyXiaoMemoryRegister 和 STonyXiaoMemoryRegister。

顾名思义，动态被动池，主要在允许动态内存申请的场合，实现服务，静态内存池，则是在某种不宜动态申请内存的场合，实现池的服务效果。上文中内存池内部的服务模块，本身就是为了预防内存碎片而存在，因此其自身逻辑显然不适合再做大规模的动态内存申请，因此笔者采用了静态被动池方式管理。

大家可以看到，出于效率起见，动态被动池，笔者使用了栈方式管理，保证了高效性。这在前文已经详细说明，此处不再赘述。

但静态被动池，大家可能会发现笔者的处理方式非常简陋，仅仅是使用了一个数据，和一个 `nUseMax` 的指针变量，即完成了管理，似乎对效率的考虑不够。前文中，我们只讨论了实现，对于效率问题也没有过多描述，因此，笔者认为有必要在此给大家详细说明一下。

7.7.3 静态被动池实施原理

静态被动池是一种比较难以实现的设计方案，因为它放弃了动态内存的申请，也就放弃了 C 和 C++ 语言大多数指针的灵活性，这给编程效率带来很大影响，而通常作为运行核心的服务池，对效率一般有较高的要求，稍有不慎，就可能大幅度影响系统的性能。笔者为此也颇费了一番苦心。

我们知道，在 C 和 C++ 中，如果不能用指针管理动态内存，一般说来，组织数据就只有用二进制缓冲区了，这在开发中，通常就是一个数组，不是字节数组，就是结构体数组，再就是对象数组。

而数组我们知道，只要涉及到检索，一定是遍历检索，而且通常是从前向后的遍历检索，这个检索的效率很低。

笔者在实现中，曾经尝试用很多方法优化这个遍历，效果都不理想。直到有一天，笔者引入概率论来思考这个问题，才豁然开朗。

我们知道，所有的静态数组，如果在装满数据的情况下，检索确实只能从前向后遍历，如图 7.1 所示：

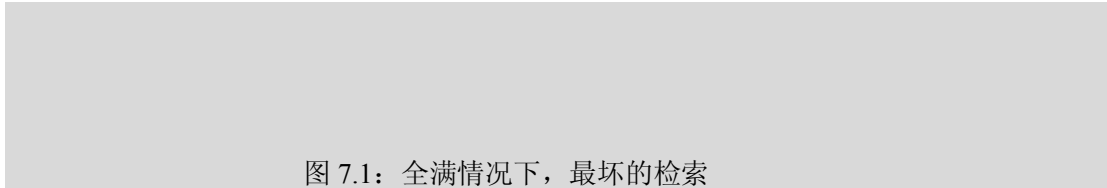


图 7.1：全满情况下，最坏的检索

这样，当我们检索的对象，是第 12 个元素时，必然会经过前面 11 次无效比较，这很浪费效率。

最坏的情况是，当我们需要检索的一个对象不存在资源池中的时候，比如，某些时候我们需要查找某个用户的连接，好转发数据给他，但这个用户在这个时间点，恰恰没有上线，socket 连接池中，没有这个用户的记录，这种查找就变得很坏。一来肯定找不到，二来系统每次必须从头至尾遍历所有数组单元，才能得到结论，这几乎直接影响很多中继服务器的转发效率。

但笔者经过思考，发现静态被动池的实际工作情形并不是这样。在池技术中，很多时候，资源是不断被重用的，即任何一个时刻，池中必然有一部分资源在等待调用，但同时，也一定有一部分资源，在外面正在为应用程序服务。

静态被动池的数组，很少有放满的时候，绝大多数时候，由于系统并发量不够大，池数组很可能只使用了前面一小部分，如图 7.2

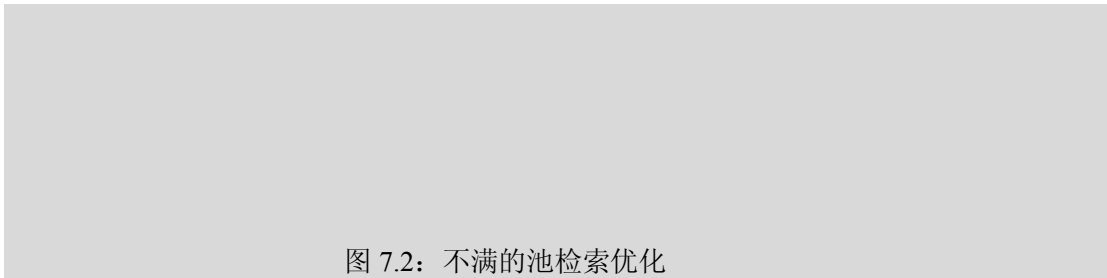


图 7.2: 不满的池检索优化

笔者发现,在这种情况下,我们如果使用一个指针 `nUseMax`,记录当前使用的最大数组单元,就可以优化全满状态下的检索效率,即每次检索,只需要检查到 `nUseMax` 即可,后续部分无需检索,因为后面的存储单元从来没有使用过。

网络服务器虽然设计能力虽然很高,但通常情况下并发量其实并不大,而池内资源是重用的,因此, `nUseMax` 通常指向并发量最大的使用情况,该数字一般远远小于系统设计能力,如一个 10000 单元的数组,通常并发量只有几百,遍历检索的范围大大缩小,因此,前文所述最坏的一种找不到现象,利用 `nUseMax` 的设计,可以很方便的实现大幅度优化。

同时,从另外一个角度,笔者发现这种设计也可以提升频繁发生的检索效率。我们知道,作为核心服务单元的被动池,内部的资源调配是很频繁的,同一个时间段,往往很多资源会不断地被应用程序申请出去使用,同时也有很多资源会被归还回来,这个时候的模型和我们平常看到的静态模型并不一样。

对于静态被动池,永远有两个关键参数,即峰值利用率和平均利用率,通常情况下,平均利用率远小于峰值利用率,二者又都远小于设计最大值。

如图 7.2,我们假定并发量是 6,即峰值利用率为 6,同时,我们假定任何时间段,平均有三个资源在外面服务,即平均利用率为 3。如果我们的检索算法是无序的话,即每次总是从数组头开始,检索到第一个可用资源,分配出去给应用程序服务,则通常情况下,数

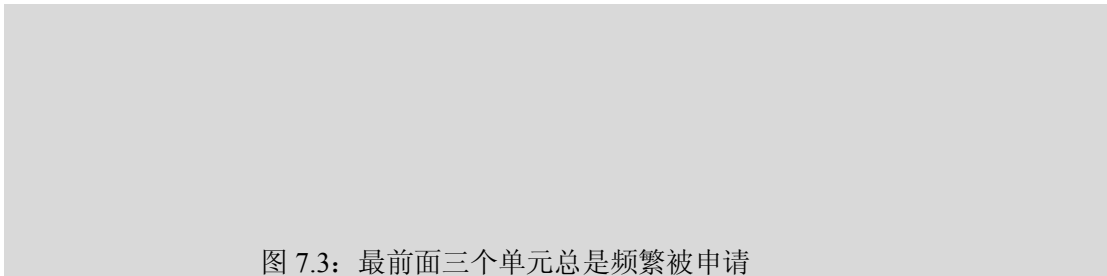


图 7.3: 最前面三个单元总是频繁被申请

组内的数据会是如下形式:

大家可以看到,当一个请求来临时,1 号单元会最先被申请出去,1 号就空缺,以此类推,第二个请求会请求走第 2 单元,等等。

但当第四个请求过来之前,第一个请求往往已经完成任务,向被动池归还了资源,如果我们的资源归还算法是寻找第一个空区放入的话,则第一个请求归还的资源,会放在 1 号单元。如图 7.4。

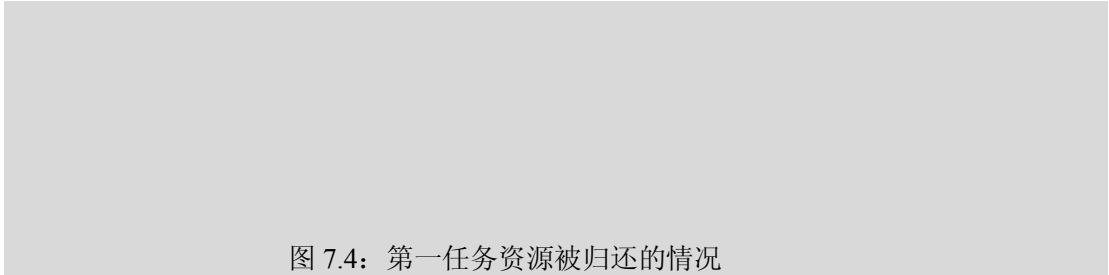


图 7.4：第一任务资源被归还的情况

这样的话，我们可以发现，当第四请求过来，申请的资源，由于是从前向后遍历，因此会将 1 号刚刚归还的资源申请走，此时，由于仅检索第一单元即获得结果，不存在遍历，效率永远是最高的。

由于上述特点，我们可以推论，静态被动池，只要按照上述原则实现检索，那在运行中，大多数情况，都是图 7.3 的形式，即 `nUseMax` 远小于设计最大值，找不到的情况下，检索效率有一个较高的水平。同时，小于 `nUseMax`，被宣告使用的区域，最前一部分的资源，总是被频繁申请和归还，这部分检索成本很低，也满足了绝大多数的高频应用需求，效率极高。

7.7.4 被动池的常见组织形式

综上所述，笔者才会如前文所述一样，使用数组和 `nUseMax` 来组织静态被动池。因此，常见的设计组织形式如下：

- 1、使用结构体定义最小的数据结构，即每个资源的数据结构。
- 2、类设计中包含一个数据结构数组，预定义一个最大值，如前文的 10000 这个值，该值根据实际需要来确定，可以做成宏的形式，方便测试阶段调整。
- 3、定义一个整型变量 `nUseMax`，专门用于记录数组使用的最大值，这也是最大并发量，该值作为所有检索遍历的终值，以此降低全局搜索成本。
- 4、Add 原则，一般是从头检索到 `nUseMax`，找到第一个空区存放数据，如果在 `nUseMax` 范围内找不到空区，则追加在 `nUseMax` 指向的末尾，同时扩大 `nUseMax` 的值。
- 5、Del 原则，一般是从头检索到 `nUseMax`，找到第一个非空的资源，提取（原存储区置空）出来交应用程序使用。

经过验证，一般以上述方式组织设计，静态被动池可以在安全和效率上取得一个平衡。

第 8 章 队列

8.1 为什么单说队列

- 8.1.1 网络同步的需求
- 8.1.2 协议信令排序的需求
- 8.1.3 存储转发的需求
- 8.1.4 异步转同步需要队列
- 8.1.5 负载均衡需要队列
- 8.1.6 等停需要队列
- 8.1.7 特例：实时转发不需要队列

8.2 几种常见的队列介绍

- 8.2.1 不是队列的队列 CBuffer
- 8.2.2 静态队列 PopBuffer
- 8.2.3 动态队列 MenQueue

8.3 动态 Buffer 类

- 8.3.1 编程思想的转变
- 8.3.2 Buffer 类的需求分析
- 8.3.3 Buffer 类声明
- 8.3.4 构造和析构函数
- 8.3.5 缓冲区大小设置函数
- 8.3.6 二进制拷贝函数
- 8.3.7 数值转换函数
- 8.3.8 二进制数据处理函数
- 8.3.9 文本字符串处理函数
- 8.3.10 数据比较函数
- 8.3.11 小结

8.4 静态 Buffer 类

- 8.4.1 类声明
- 8.4.2 构造函数和析构函数
- 8.4.3 缓冲区设置函数
- 8.4.4 二进制拷贝函数
- 8.4.5 数值转换函数
- 8.4.6 二进制数据处理函数
- 8.4.7 文本字符串处理函数
- 8.4.8 数据比较函数
- 8.4.9 小结

8.5 PopBuffer

- 8.5.1 PopBuffer 基本需求分析
- 8.5.2 基本数据结构介绍
- 8.5.3 基本类模型
- 8.5.4 构造函数和析构函数
- 8.5.5 工具服务函数
- 8.5.6 业务查询函数
- 8.5.7 添加 AddLast
- 8.5.8 提取 GetAndDeleteFirst

- 8.5.9 MoveAllData
- 8.5.10 PopBuffer 小结
- 8.6 MemQueue
 - 8.6.1 动态队列的管理原则
 - 8.6.2 基本数据结构介绍以及优化考虑
 - 8.6.3 基本功能类声明
 - 8.6.4 构造函数和析构函数
 - 8.6.5 辅助工具函数
 - 8.6.6 追加 AddLast
 - 8.6.7 提取 GetAndDeleteFirst
 - 8.6.8 PopBuffer 相关操作
 - 8.6.9 文件保存相关操作
 - 8.6.10 线程安全锁封装类
- 8.7 关于队列的小结

第 8 章 队列

队列可以说是商用并行工程中最重要数据结构，有太多的地方用到队列。从简单的数据传输组织，到任务池最核心的任务组织，队列几乎无处不在。笔者在此，为大家介绍商用数据传输工程中，常见的各种队列形式。

8.1 为什么单说队列

我们在大学中，通常会学到很多数据结构，但在本书涉及的商用并行工程范围，队列是最常用的数据结构。这其中主要的原因是，队列有“先进先出”的逻辑特点，即一个业务，一笔交易，一条信令，从其产生开始，在整个传输过程中，其顺序不会打乱，这符合大多数业务模型的需求，因此用途极广。

8.1.1 网络同步的需求

我们知道，服务器集群和分布式数据库系统中，存在大量的网络数据同步需求，而这类同步需求，一般都有信令顺序要求，这势必需要队列来完成。

例如，在一个银行系统中，一个客户帐户为 0 元，他先存进 1000 元钱，然后取出 500 元钱，这在实际生活中，是经常发生的，也是正常的。

但是，如果银行柜台和中心数据库的传输系统，无法保证信令的顺序，则很可能变成先取 500 元，再存 1000 元，这样，当用户取钱时，由于帐户为空，服务器肯定拒绝交易，本次交易失败，这就是事故了。

因此，大多数集群系统，设计时都要求使用队列来保证信令的顺序。

8.1.2 协议信令排序的需求

我们前文说过，在很多网络数据传输场合，都有协议的存在，也就出现了信令。很多时候，出于降低开发成本的目的，信令的约定一般都很简单，并且，存在信令重用的现象。这就造成了接收端对信令的理解，很多时候要参考上下文，即需要参考前导信令来判断当

前信令的真实含义。此时，如果信令的传输顺序发生了变化，就会造成理解错误，导致传输无法进行。

例如，我们在通信中经常使用 Ack 信号来表示连接握手成功，但随着使用场合的不同，Ack 信号被赋予了太多的含义，可以是一次请求，不需要回应，对方返回 Ack 表示通信结束，也可能是通信的双方一次简单的心跳行为，维护链路，还有可能是某次确定传输之后，接收方返回接收正常的信号。等等。这在各种协议中非常常见。

这说明，这类协议系统的数据传输，一定是顺序的，乱序的传输，最终会导致 Ack 含义完全错误的理解，协议握手最终会失败，通信无法继续。

8.1.3 存储转发的需求

我们目前使用的互联网是基于 IPV4 协议，并不是 IPV6，因此，对于全世界来讲，公网的 IP 地址总的来说是有限的，大约只有 40 亿个左右，这直接造成了目前公网 IP 紧张的局面。

人们为了解决这个问题，就发明了路由器等 NAT 设备，通过把局域网内的设备，共用一个公网 IP 上网的方式，来解决 IP 资源不够的问题。这在生活中很常见。

不过这就带来一个问题，就是位于两个内网中的用户，如何利用互联网通信。由于大家都是共用公网 IP，也就没有独享的 IP 地址，另一方很难直接发起对对方的直接连接请求，也就无从通信。

通常的做法，是在公网架设中继服务器，由于其本身具有公网 IP，可以响应直接连接请求，因此每个内网客户，可以先连接这个中继服务器，再把要转发的信息，由这个中继服务器转发给通信的另一方，如图 8.1 所示。



图 8.1：两个内网的计算机，通过中继服务器通信

如图所示，当 A 试图直接连接内网 2 的 B 时，由于 NAT 的特性，在没有管理员做静态映射的前提下，这种连接一般都会被内网 2 的 NAT 拒绝，传输无法成功。

只有当公网设置中继转发服务器，A 和 B 同时都连接到中继服务器时，传输才会成功（图中虚线模拟传输行为）。

当然，P2P 是另外一个话题，即通信双方使用 UDP 协议，通过 NAT “打洞”，双方在某种情况下还是有可能直接通信的。但，就算是 P2P 通信，在通信监理时，仍然需要一台中心服务器协助握手，方能正确传输。

大家可以想象一下，这个中继转发服务器，是不是必须保证传输的顺序，因为它是一个纯透明的传输层，是没有权利修改数据的传输顺序的，否则就可能影响业务层。因此，它必须绝对保证收发顺序一致。

这里，就需要用到队列。

8.1.4 异步转同步需要队列

我们知道，在商用服务器的设计中，通常面临的客户连接请求，是并发的，也是不可预测的，但是，一个服务器，即使是并行系统设计，但总的服务能力，比起客户需求来说，总是有限的。

通常情况下，我们需要把异步到来的服务请求，放到一个队列中，实现串行化，由核心服务模块一条条地处理，再分别发送回应。这个异步转同步的过程，必须用到队列。

提示：虽然有时候看起来客户请求没有先后顺序需求，但仍然不能用栈来解决，栈的特点“后进先出”，总是最后的请求最先得到响应，当任务繁忙到一定程度，就可能有些请求，被长期压在栈底，无法获得响应，最终导致客户请求的超时失败，且栈中如果积压大量的无效过期请求，也极大地浪费了服务器资源，甚至引起故障。因此，异步转同步，一般说来，都是用队列解决。

8.1.5 负载均衡需要队列

前文我们讲过，服务器的服务能力，比起无穷无尽的客户请求来说，总是偏小的。所有的商用服务系统，一般都有个最大峰值负载和平均负载的参量，后者往往远小于前者。

我们这里强调商用服务系统，就是指由企业投资架构，在公网实现商业运营服务的网络服务设备，既然是商业运营，就一定有成本和利润的考虑，我们设计一个系统，如果都是以最大峰值负载来设计，显然没有必要，因为很可能一年里面，只有一两次，负载能达到这么大的量，其他时候，都是平均负载的业务量。

但这就带来一个问题，如果我们按照平均负载设计系统，万一发生了峰值负载怎么办？总不能立即挂死，拒绝服务吧。

这里就需要用到队列。

提示：常见的做法，是前级接收客户请求的任务模块和后端处理请求的模块分流，二者是两个模块，它们之间设置一个比较大（容量起码超过最大并发量）的队列作为缓冲，前级只管接收请求，放入队列，后端服务模块不断从队列中提取请求并处理。这样当某个时刻，峰值压力过大，请求过多的时候，前级收到请求立即入任务队列，马上返回来处理下一连接请求，对外能提供持续服务能力，而后端服务能力不足，最多表现在队列淤积的请求有所增加，队列长度开始变长，但在一定超时范围内，这都是可以允许的。但整体看来，这台服务器的服务并没有终止，且连绵不绝地提供服务。

这实际上是利用队列，分摊了峰值情况和平均情况下的负载，利用队列的缓存功能，把某些请求的处理时间延长，进而降低服务器压力，使服务器能实现平稳服务的目的。

8.1.6 等停需要队列

在商用传输类工程中，通常有两大类传输请求，一类是确定传输，一类是不确定传输。前者一般用于关键控制信令的传输，要求传输信令不能丢失、乱序，传输层需提供失败重传机制，后者则通常用于流媒体等传输环境，允许一定的丢包率，利用业务层编码和协议来弥补丢包的损失。

在确定传输中，有一种很常见的设计就是等停，即发送方发送完毕，必须等待接收方的一个 ack 信号，表示收到，才会发送下一条，如果没有收到，超时失败，表示本轮发送失败，则必须重传。

在公网的数据传输中，由于要面临各种情况，确定传输的时间成本很高，其传输速率一般远小于数据到来的速度，因此，中间必须使用队列进行缓冲。

当然，还有一种情况，更加说明了队列的重要性，在某些安全性要求较高的服务器集群中，需要承诺用户信令保留多少天不会丢失。这是考虑到，当某种异常情况下，传输的目标网络中断，需要传递的信令必须在中继服务器上做长时间保留，这个时候，更加需要队列的帮助。

8.1.7 特例：实时转发不需要队列

当然，还有一种情况，即某些流媒体，实时转发的场合，由于传输信息有很强的时效性，超过时效的数据传输没有意义，可以抛弃，这时候一般不需要队列，而是通过锁方式抢夺连接 socket，做直接转发操作。

因此，很多实时转发场合，反而不使用队列。

8.2 几种常见的队列介绍

队列在商用并行工程中，使用面很广，根据笔者的经验，主要有静态队列，动态队列和文件型队列三种。虽然很多应用程序框架都提供了队列模块供程序员使用，但出于效率考虑，笔者一般倾向于自己开发。

提示：本书主要论述基于内存的静态队列和动态队列，文件型队列由于涉及很多具体业务开发，本书暂时不做详细论述。

队列最核心的功能，就是允许应用层添加随机长度的数据，同时按添加次数分条记录，并能按照条目，逐一从队列头弹出使用。这个条目，笔者通常称之为元素。

原则上，队列最重要的工作接口，有以下几个：

- 1、AddLast，这是将一笔数据添加到队列末尾
- 2、GetFirstLenght，获得第一笔数据的长度，这是为了应用程序准备缓冲区考虑
- 3、GetFirst，获得第一笔数据
- 4、DeleteFirst，删除第一笔数据
- 5、GetAndDeleteFirst，提取第一笔数据（带删除，相当于 3 和 4 的结合）

8.2.1 不是队列的队列 CBuffer

这个是笔者在工程中常用来描述二进制数据的缓冲区类，主要做传输收发使用。不过，在某些情况下，为了某些信令拼接的方便，笔者为其添加了很多队列相关的特性，因此这

里也列举出来作为示例。同时，下列很多队列的操作，也包含针对该对象的操作特性，为了方便起见，也有必要提前做个介绍。

该 Buffer 类主要依托内存池工作，内部保存一段二进制数据，并且可以很方便地插入、提取其中部分数据，实现某些信令的拼接和拆解工作。本章将在后文中详细介绍。

8.2.2 静态队列 PopBuffer

静态队列在商用并行工程领域特别有用。简单说，就是使用一个静态的二进制 buffer，在其内部实现队列逻辑管理，方便对二进制缓冲区的操作。

前文我们说过，在很多确定传输的场合，传输报文的速度，远远低于信令产生的速度，根据笔者经验，一般带 Ack 的确定传输，每秒钟最多发生 10~20 笔交易，也就是平均 50~100ms 一次成功的传输行为。这个速度在网络服务中，是慢得不可想象的，服务器的效率极其低下。尤其是同时服务多个客户端的请求响应服务器来说，更加不可忍受。

笔者曾经专门分析其原因，发现主要的时间成本，出现在 TCP 连接建立时的成本，以及发送方发送完毕后，等待接收方回传 Ack 信号的等待成本。

换言之，每一次发送的成本很高，但是，每次发送中，发送有效业务数据的时间并不长，因此这里带出一个话题，“**商用数据传输工程中，信令应该打包发送。**”

即每次准备好发送时，发送方不应该仅仅发送一条信令，而是应该考虑，将当前队列缓冲区内容，全部打成一个包发送，接收方收到后，再拆解，这样可以大幅度提升传输效率，实现高效传输。

而经过笔者分析，发现利用队列格式来打这个信令包，是最合适的，因为很直观，接收方拆包也很容易。

因此，在笔者的传输程序中，经常会使用下面介绍的 PopBuffer 类来做信令的打包和拆包工作。

这种用法时，PopBuffer 基本上属于前文所述的“粘合类”方式，即对象本身没有数据缓冲区，而是发送程序准备好一个信令报文缓冲区，把这个缓冲区“赋给”PopBuffer 作为队列缓冲区。之后就可以不断调用 PopBuffer 的 AddLast 来添加信令，直到缓冲区满，或者没有更多新信令了，然后开始发送。

当接收方收到一个二进制报文后，也将其“赋给”一个 PopBuffer 对象，然后不断调用 GetAndDeleteFirst 方法，提取信令处理，对于信令集方式传输，这大概是最简单的操作模型了。

由于这个原因，静态队列 PopBuffer 类成为使用率非常高的队列之一。

8.2.3 动态队列 MenQueue

当然，PopBuffer 缓冲区，由于使用线性编址的静态队列，本质上还是二进制数组，因此，在 DeleteFirst 动作时，需要将后续数据向前 Move，使用成本还是很高的。除了报文编组、信令打包这类特殊场合，一般不建议使用。

在笔者经常使用的异步转同步缓冲队列，工程系统各种主队列中，为了保持效率，主要使用依赖内存池的动态队列来完成。

这类队列内部大量使用指针管理动态内存块，利用指针变换获得很高的效率，因此也比较常用。

8.3 动态 Buffer 类

Buffer 类，在笔者的工程库中，定名为 CTonyBuffer，这主要是因为该命名是一个通用命名，极易与其他同事的类似模块重名，为了避免以后整合的风险，特地使用笔者的英文名，做了定名修饰。

8.3.1 编程思想的转变

这个 Buffer 类其实构建很晚，是笔者近一两年构建的模块产品。笔者设计这个模块的动机其实还有一段故事。

8.3.1.1 纯静态编程思想

大家可能听说过，近年来，印度的软件外包行业发展很迅速，远远超过了中国大陆。笔者也阅读过很多分析性文章，其中，有几点让笔者很惊奇：

1、印度的程序员产量很低，一个程序员一个月，大约也就几百行代码，他们主要的工作是文档，文档化交互，是他们工作的主体。

2、印度的程序员，都很“笨”，他们不会使用什么高深的动态内存分配，链表，树等复杂数据结构，很多时候，他们的产品，就是一个简简单单的数组，然后就把产品做出来了，这自然带来了软件绝大的安全性，产品质量很好。

3、印度程序界，架构师是架构师，程序员是程序员，分工非常明确，公司中层级划分也很严重，每个人都有一定的权限，程序员严格按照设计师的文档来工作，决不多做或者少做。

笔者算是长期经历过中国的软件开发现状的程序员，应该承认，上述几条，在现代软件工程开发管理的视角看来，确实是很大的优势。

就笔者所知，中国程序员，每个人都是设计师，没人承认自己是软件蓝领。对于客户需求分析和系统设计，缺乏必要的尊重，甚至根本无系统分析可言，经常是几个人一商量，就开始做程序，这样的程序，质量怎么能稳定。

最重要的是，包括笔者自己，中国的程序员很喜欢玩一些技巧性的东东，在我们的设计中，大量使用链表，树等动态设计，经常感到，中国的程序员，更像艺术家，而印度的程序员，更像工程师。

笔者也就是在这个时期，开始深入思考商用工程设计，需要什么样的程序员。

笔者认为，在商用工程领域，需要的不是少数的艺术家，专门制造精品的大师，商用工程程序员，以盈利为优先，以满足客户需求为己任，我们更多的需要的是，没有个性的工程师，严格，甚至死板遵守设计规范，以恒定质量输出产品的软件工程师。太多的个性，在团队工作中，无法保证沟通效果，也无法保证产品质量，非但无益，反而有害。

因此，笔者在很长一段时间，软件设计思想趋于保守化，所有的程序设计，建立在严格的数据边界界定，以及详细的系统设计之上，在程序中，偏向于使用数组等静态元素，大家可能能感觉到，笔者的很多工程库模块设计，都提供静态和动态两种方式，这就是这段时间思想的遗迹。

这样显然带来了很大的好处，笔者的程序质量大大提高，产品的 bug 很少。

8.3.1.2 多线程环境的特殊性

静态数组设计固然很好，可以很大程度地保证程序的安全性，但是，也有其弱点。

以数据传输为主的商用并行工程，涉及到很多 7*24 小时服务器和嵌入式设备的开发，这些设备，要么像服务器，内部同时运行多个服务，要么像嵌入式，内存等资源很少，因此，笔者的程序在运行期间，出于成本考虑，获得的实际运行资源很少。

而笔者习惯性地采用静态数组方式开发，很显然是一种浪费空间的举动，这样的设计，虽然能完成开发任务，设计出很稳定的产品，但是，资源消耗过大，成本偏高，不能算最好的设计。

这样的设计，还有一个致命的隐患。我们知道，静态数组一般存在于程序运行期的栈空间，而多线程环境下，由于各个线程生命周期不一致，不能共同使用同一个栈空间，以免发生内存混乱。操作系统一般为每个线程独立开辟一个栈空间，互不干涉。这实际上表



图 8.2：多线程程序内存示意图

明，多线程环境下，进程内部有多个栈空间，分别为不同的线程服务。如图 8.2

我们知道，一般函数的浮动栈，主要用于函数的调子活动，即一个函数 A，Call 另外一个函数 B 时，会为其开辟一个内存空间，放置函数 B 内部变量等数据。当 B 返回时，拆除该空间，其内部数据宣布作废。

由此我们可以得知，其实操作系统调用一个 C 语言程序，从 main 开始，逐级调用各个函数完成功能，实际上构成了一个函数调用链，这个链上记录了函数的层级调用关系。当程序的功能展开，随着调用函数的增多，这个链开始变得很长，同时，其对应的浮动栈，使用量开始增加，当功能逐渐收拢，函数层级返回，链变短，浮动栈的使用也开始减少，最后，当程序完成所有的功能，回到链头 main 函数，返回，所有浮动栈拆除，操作系统拆除进程空间，本次执行结束。

对于一个单任务的程序，上述过程，就是一个完整的执行生命体，在这次执行生命期间，同一时间段，只会调用一个函数处于栈顶，并被激活运行，因此，系统只需要准备一个浮动栈就足够了。

但是，在多任务环境下，由于有多个线程并行，上述模型并不准确，由于每个线程，都有自己独立的时间片，相当于独立的执行生命体，也就有了自己独有的函数调用链，如果所有的函数共用一个浮动栈空间，则会造成很大混乱，甚至可能由于多线程争用问题，直接导致程序崩溃。

因此，C 和 C++多线程库，为每个线程准备了独立的浮动栈空间，如图 8.2 所示，一个进程的内存空间中，有多少个线程在运行，就有多少个浮动栈。这个线程使用的浮动栈，通常叫做线程栈。

这就带来了一个问题，既然是多个执行生命体并发使用同一块内存，那大家使用的内存就不能太大，免得互相干扰。必须为其规定一个最大大小。一般说来，这个线程栈大小，可以通过编译器的参数设置，默认情况下，Windows 下 VC 编译器一般默认为 1M，Linux 下 gcc 编译器一般默认为 10M，嵌入式系统比较复杂，笔者也没有可以参考的数字，各位读者有兴趣的话，可以自行研究。

我们以 Windows 开发为例，当我们开启一个线程，这个线程开始执行功能，不断调用函数，当函数调用链最长的时候，就是调用层数最深的时候，所有这条链上的函数，内部变量所占用空间之和，不能超过 1M，否则会失败，比较可怕的是，这种失败通常是直接崩溃，系统不给原因。

提示：各位读者要是有兴趣，不妨在 Windows 下开启一个线程，在该线程内部定义一个 2M 左右的数组，就可以看到立即崩溃的景象了。

这里笔者给大家看一下函数实例：

```
void Func(void)
{
    char cArray2[748*1024];          //定义一个 748k 的数组
    //...
}
void Thread(void)
{
    char cArray1[512*1024];          //定义一个 512k 的数组
    Func();                          //调用 Func，函数调用链+1，栈层+1
    //...
}
```

我们假定这是一个 Windows 程序的一段线程函数，Thread 函数内部声明了一个 512k 的数组，同时，在其调用的 Func 函数内声明了一个 748k 的数组。这段代码看起来没有任何问题，但运行时会崩溃。

原因很简单，VC 编译器默认线程栈只有 1M，当线程开始运行时，cArray1 占用了 512k，这没有超限，程序可以正常运行，但是，当 call 到 Func 这个函数时，此时 Thread 函数没有退出，cArray1 占用的空间没有释放，因此，cArray2 会在 512k 的基础上，再占用 748k 空间，超过了 1M 的限额，程序崩溃。

最要命的是，这个崩溃通常会发生在汇编语言阶段，即这个 Func 函数还没有正式创建时崩溃，利用 VC 的调试器，看不到崩溃的点，因此，很难查找 bug。

而通常情况下，我们知道，C 和 C++无错化程序设计，要求每个函数写得很简单，反过来，这样写出的程序，函数调用链一般很长，浮动栈也很深，程序员凭肉眼，很难判断出哪一个分支的调用链，所有函数内部变量之和超出了限额，会导致崩溃，因此，这类 bug 也是无法跟踪的 bug，应该予以避免。

正常情况下，每个函数，都只有几个临时变量，这占用的空间很小，但根据笔者的经验，以下的情况应该特别注意：

1、函数内部实例化了本地对象，而不是使用 `new` 在远堆动态创建的情况，要检查对象内部是否有大数组，如果有，尽量改写成指针模式，在构造函数 `malloc`，在析构函数 `free`，否则，可能崩溃。

2、每个函数内部最好不要声明数组，数组放到全局，或者放到类中声明。

3、对象的创建，最好都使用远堆创建，尽量不好静态实例化。

4、特别关注线程是否调用了太深的函数链，必要时，可以考虑分割线程，以双线程同步等待，实现单线程的功能，借此扩展函数栈。

5、必要时，可以考虑调整编译器参数，扩大线程栈空间大小，但这条一般不推荐。

8.3.1.2 动静结合的编程思想

大家可以看到，上述几条原则，和我们前面说理念，有很多矛盾之处，不过这没有办法，C 和 C++ 无错化程序设计方法，也仅仅是一个通理，在某些特定场合，由于系统的限制，其思路也必须要调整。

正是因为笔者近年来深入思考多线程并行开发环境，感觉到上述限制因素，这才强行扭转了静态编程思想，逐渐回到动态编程思想上来。

但是，我们也要承认，动态编程，也有其优缺点，不能一概而论，因此，笔者不忌讳使用动态内存申请等手段，开发时，有时候也会在函数或类内部申请大型数组，但这都是基于对总内存消耗量，函数调用链的把握基础上进行的，随时关注这类潜在的系统限制，并及时调整算法，这，大概就算是动静结合的编程思想了。

而 `Buffer` 这个模块，基本上就产生于这个思想。在内存池的帮助下，动态内存申请不再可怕，内存碎片的影响也降低到一个很低的程度，因此，笔者在设计中，开始敢于使用动态内存申请。

而 `Buffer` 类，就专门应对商用数据传输工程中，对于信令、报文的缓冲区需求，以动态内存代替原有的静态二进制数组，以此避免上述线程栈的潜在影响。

8.3.2 `Buffer` 类的需求分析

`Buffer` 类由于被笔者赋予了太多的责任，要负责商用数据传输工程中，所有的信令构建、报文拼接，因此，笔者将其设计为一个纯粹的工具类，方便后续程序使用。

并且，这个类笔者并没有将其定型，而是始终保持不断维护和开发之中，有了新的需求，不断添加方法。这在工程库维护中，虽然不太常见，但是也可以理解，太通用的类，很可能处于不断维护中，是个长期演进的过程。

值得一提的是，`Buffer` 类并没有做任何线程安全锁保护，这是考虑到其主要工作于函数调用链上，处于一个线程中，无需考虑跨线程安全。当然，以后如果有需求，需要对其作安全防护，基于资源锁的理念，笔者也可以很方便地为其添加一个封装类，实现线程安全。

`Buffer` 类的核心需求，其实就是封装一段动态内存区，以及其长度参数，提供各种方法，帮助应用层方便地存入和提取数据，有点类似于 MFC 的 `CString` 类，但是，范围更广泛，不仅仅支持字符串，也支持二进制数据。最关键的是，`Buffer` 提供该动态内存区的安全访问机制，析构函数自带释放，避免了内存泄漏。

这是由于网络协议通常包含二进制信令和文本字符串信令两种，各有优缺点，因此，需要二者都兼容。

在网络传输中，`Buffer` 类有个很大的功能，也很好用。很多时候，网络传输使用的是应答模型，即客户端对服务器，采取一问一答的对话方式，这在静态数组方式开发中，就很麻烦，因为很多时候，客户端仅仅知道请求的长度，但服务器回应报文的长度，并不知道，为了避免越界，就只有两种选择：

1、接收函数动态申请一块内存，保存回应报文，返回给上层应用层，应用层解析完毕，再释放该内存，但这不符合“谁申请，谁释放”原则，显然不好。

2、接收缓冲区由上层准备，以参数形式传入接收函数接收，此时需要预测回应报文的长度，通常是规定一个很大的数字，比如 200k，这显然容易造成上述的栈空间溢出，也不好。

此时，如果使用 **Buffer** 类，由于该类内部的缓冲区是动态的，自然可以根据回应报文自动调整，且都是在堆内存申请，不会导致栈空间溢出的现象，该问题迎刃而解。

笔者在实做时，甚至将就使用请求的缓冲区，直接存放回应返回，程序开发更加方便。

在设计时，由于 **Buffer** 类包含了很多二进制报文拼接的方法，如追加到最后，插入到开始等，无形中，也包含了一定的队列特征，可以视为一种二进制，没有条目约束的队列。

8.3.3 Buffer 类声明

Buffer 对外表现为一个类对象，方便应用层操作。由于这个类依赖内存池进行操作，因此工作前必须将内存池指针导入到构造函数。

```
//Buffer 类
class CTonyBuffer
{
public:
    //构造函数，注意传入 pMemPool 内存池对象指针。
    CTonyBuffer(CTonyMemoryPoolWithLock* pMemPool);
    ~CTonyBuffer();
public: //请注意，典型的工具类特征，所有内部变量，全部公有，方便应用层调用
    CTonyMemoryPoolWithLock* m_pMemPool;    //内存池指针
    char* m_pData;                          //动态内存缓冲区指针
    int m_nDataLength;                      //内存缓冲区长度（代数据长度）

public:
    //////////////////////////////////////
    //尺寸设置函数
    bool SetSize(int nSize);                //设置新的大小
    bool InsertSpace2Head(int nAddBytes);   //在前面插入空白
    bool AddSpace2Tail(int nAddBytes);      //在后面插入空白
    void CutHead(int nBytes);               //从前面剪切掉一段数据
    void CutTail(int nBytes);               //从后面剪切掉一段数据

    //////////////////////////////////////
    //数值转换函数
    bool SetInt(int n);                     //将一个整数以二进制方式拷贝到缓冲区，带网络字节排序
    int GetInt(void);                       //以整数方式获得缓冲区的数值
    bool SetShort(short n);                 //将一个短整数以二进制方式拷贝到缓冲区，带网络字节排序
    short GetShort(void);                   //以短整型方式获得缓冲区的数值
    bool SetChar(char n);                   //将一个字节以二进制方式拷贝到缓冲区
    char GetChar(void);                     //以字节方式获得缓冲区的数值
```

```

////////////////////////////////////
//二进制数据追加函数
//追加数据到最后，返回新的数据长度
int AddData(char* szData,int nDataLength);
//插入数据到最前面，返回新的数据长度
int InsertData2Head(char* szData,int nDataLength);

```

```

////////////////////////////////////
//二进制数据拷贝函数
//拷贝到一块目标缓冲区，受传入的缓冲区长度限制
int BinCopyTo(char* szBuffer,int nBufferSize);
//从一块来源缓冲区拷贝数据到本对象中
int BinCopyFrom(char* szData,int nDataLength);
//从另外一个 Buffer 对象拷贝数据到本对象
int BinCopyFrom(CTonyBuffer* pBuffer);

```

```

////////////////////////////////////
//文本数据拷贝构造函数
int StrCopyFrom(char* szString);
int Printf(char* szFormat,...);

```

```

////////////////////////////////////
//数据比较函数
int memcmp(char* szData,int nDataLength);
int strcmp(char* szString);
};

```

8.3.4 构造和析构函数

Buffer 类的构造函数和析构函数相对比较简单，但需要关注的是，需要仔细控制内部指针和长度参数的值，这个 Buffer 的设计逻辑，内部缓冲区纯动态申请，没有数据，就没有缓冲区，指针为空，有数据的话，缓冲区则和数据长度刚好相等，以尽量节约内存空间。

```

CTonyBuffer::CTonyBuffer(CTonyMemoryPoolWithLock* pMemPool)
{
    m_pMemPool=pMemPool;          //保存内存池指针
    m_pData=null;                  //注意此处，动态内存块指针并未分配，先保持空
    m_nDataLength=0;              //长度也保持为 0
}
CTonyBuffer::~CTonyBuffer()
{
    SetSize(0);                   //SetSize 为 0，表示清除动态内存块，下文有介绍
}

```

8.3.5 缓冲区大小设置函数

缓冲区大小设置函数，主要包含从调整缓冲区大小，从前后插入空白，从前后剪切长度等函数。

8.3.5.1 SetSize

大小设置函数，最重要的就是 SetSize 函数，笔者把几乎所有大小相关的功能都聚合到其中完成。本函数破坏原数据。

```
//由于动态内存申请，可能会失败，因此返回 bool 量，成功为真，失败返回假
bool CTonyBuffer::SetSize(int nSize)    //给定新的大小
{
    if(!m_pMemPool) return false;        //防御性设计，如果内存池指针为空，无法工作
    m_nDataLength=nSize;                 //先给缓冲区长度赋值
    if(0==m_nDataLength)                 //注意析构函数中的 SetSize(0)，此处处理
    {                                     //如果设置大小为 0，表示释放缓冲区
        if(m_pData)
        {
            m_pMemPool->Free(m_pData);    //内存池释放
            m_pData=null;                 //释放后立即赋初值 null，避免下文误用
        }
        return true;                     //这也是操作成功，返回真
    }
    //此处开始，新设置的缓冲期长度，一定不为 0
    if(!m_pData)                         //如果原指针为空，表示没有数据，需要 Malloc
    {
        //请注意这里对内存池 Malloc 函数的调用，这也算是内存池的应用实例
        //内存池申请的指针都是 void*，这符合 C 语言 malloc 的规范，
        //而本类的指针都是字符串型，因此需要做强制指针类型转换
        m_pData=(char*)m_pMemPool->Malloc(
            m_nDataLength,                //申请的内存块长度
            "CTonyBuffer::m_pData");      //注意，这里特别声明申请者身份
                                           //一旦发生内存泄漏，本对象退出时忘了释放
                                           //内存池的报警机制即会激活，打印这个字符串
        if(!m_pData)                     //请注意，这里二次判断，是判断申请是否成功
        {
            m_nDataLength=0;              //没有成功，则把长度赋为 0
            return false;                  //申请失败，返回假
        }
        else return true;                 //成功就返回真
    }
    else    //这是原有 m_pData 不为空，就是已经有一个内存区的情况
    {
        //使用 ReMalloc 函数做内存区域再调整，默认，拷贝原有内容
        //请注意，这里是内存池 ReMalloc 功能的实例，将原有指针传入
        //经过 ReMalloc 修饰后，返回新的指针，注意强制指针类型转换
        m_pData=(char*)m_pMemPool->ReMalloc(m_pData,m_nDataLength);
        if(!m_pData)
        {
            m_nDataLength=0;              //申请失败，返回假
            return false;
        }
        else return true;                 //成功返回真
    }
}
```

8.3.5.2 InsertSpace2Head

InsertSpace2Head 函数，是在原有缓冲区头部插入一段空间，方便前导信令的插入。大家请注意函数定名的表意性。本函数不破坏原数据，但原数据会向后 **Move**，为前导空区留出空间。

```
//在前面插入空白，同上，动态内存申请可能失败，所以返回 bool 量
bool CTonyBuffer::InsertSpace2Head(int nAddBytes)
{
    bool bRet=false;                //预设返回值
    int nNewSize=0;                 //新的空间大小变量
    char* pBuffer=null;             //先定义一个新的二进制缓冲区指针
    if(!m_pMemPool)                 //防御性设计，防止内存池为空
        goto CTonyBuffer_InsertSpace2Head_End_Process;
    nNewSize=m_nDataLength+nAddBytes; //求得新的长度
    //请注意这里，申请一个临时缓冲区，其长度等于原长度+增加的长度
    pBuffer=(char*)m_pMemPool->Malloc(nNewSize,
        "CTonyBuffer::InsertSpace2Head():pBuffer"); //请注意这段说明文字
    if(!pBuffer)                    //缓冲区申请失败，跳转返回假
        goto CTonyBuffer_InsertSpace2Head_End_Process;
    //此处为防御性设计，如果原有缓冲区为空，则不做后续的拷贝动作
    if((m_pData)&&(m_nDataLength))
    { //这是将原有缓冲区内容拷贝到临时缓冲区后半部分，与新增字节构成一个整体
        memcpy(pBuffer+nAddBytes,m_pData,m_nDataLength);
    }
    //当所有准备工作完成，调用本对象的二进制拷贝函数，将临时缓冲区内容拷贝回本对象
    //二进制拷贝函数，后文有介绍，当然，其拷贝成功与否，也作为本函数的返回值
    bRet=BinCopyFrom(pBuffer,nNewSize);
CTonyBuffer_InsertSpace2Head_End_Process:
    //不管是否成功拷贝，释放临时缓冲区
    if(pBuffer)
    {
        m_pMemPool->Free(pBuffer);
        pBuffer=null;
    }
    return bRet;                    //返回操作结果
}
```

这里特别请大家关注一个细节，本函数的设计看似比较繁琐，简简单单的一个前方插入，动用了一次动态内存分配，使用临时缓冲区做了多次拷贝动作，当拼接完成时，才会最后动用本对象的二进制拷贝函数，拷回本对象。看起来非常麻烦，效率也不高。不过，请大家注意，这是必须的。

我们知道，**memcpy** 是我们常用的一个拷贝函数，也是 C 语言基本库内的功能函数，所有的程序员都在使用。但有个细节恐怕不是每个程序员都能关注。就是内存拷贝函数的“从前拷贝”和“从后拷贝”问题。

请大家先看这两段拷贝函数：

```
//从前拷贝函数
void CopyFromHead(char* pD,char* pS,int nSize)
{
    int i=0;
    for(i=0;i<nSize;i++)          //循环是从头到尾
    {
        *(pD+i)=*(pS+i);
    }
}
//从后拷贝函数
void CopyFromTail(char* pD,char* pS,int nSize)
{
    int i=0;
    for(i=nSize-1;i>=0;i--)      //循环是从尾到头
    {
        *(pD+i)=*(pS+i);
    }
}
```

我们再来看这个例子，这段例子中，我们准备了一个 20 个单元的数组，并在里面填充上 0~19 的数字，然后，我们希望把数组中的数字，向后平错 5 个单元，即从 5 号单元开始，填入 0~14 之间的数字。代码如下：

```
void CopyTest(void)
{
    char cArray[20];          //准备 20 个单元的字符串数组
    int i=0;
    for(i=0;i<20;i++)
    {
        cArray[i]=i;          //给数组赋初值，0~19
        printf("%d ",cArray[i]); //打印数组原始内容
    }
    printf("\n");
    // CopyFromHead(cArray+5,cArray,15); //选择从前拷贝
    // CopyFromTail(cArray+5,cArray,15); //选择从后拷贝
    for(i=0;i<20;i++)
    {
        printf("%d ",cArray[i]);
    }
    printf("\n");
}
```

这中间，我们有两个选择，一个是从前拷贝，一个是从后拷贝，我们来看看这个效果。

1、从前拷贝的结果

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
00 01 02 03 04 00 01 02 03 04 00 01 02 03 04 00 01 02 03 04
      ^请注意，从此处出现错误
```

2、从后拷贝的结果

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
00 01 02 03 04 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14
```

大家注意到，从前拷贝出现了 **bug**。我们本意是获得后面这个结果，即从第五单元开始，获得连续 0~14 的数字，但是，从前拷贝的结果，显然不对。

究其原因，由于本例拷贝目标区与数据来源区重合，拷贝动作本身，影响了来源，即当我们把第 1 单元的数字拷贝到第 5 单元时，已经破坏了将来要拷贝到第 10 单元的数字 5，导致结果异常。

这在汇编语言中看得比较明显，但在 C 语言中，由于 `memcpy` 一般屏蔽了拷贝的顺序细节，程序员很难区分。并且由于标准库的 `memcpy` 并没有规定一定是从前拷贝还是从后拷贝，因此，对于上述从缓冲区头向后 `Move`，一般笔者必须使用一个临时缓冲区来存储中间拼接状态，以保证绝对的拷贝安全。

8.3.5.3 AddSpace2Tail

`AddSpace2Tail` 函数，是在原有缓冲区后部，追加一段空白，这个操作相对比较简单。实际上完成的就是 `SetSize` 的逻辑。本函数保留原数据。

```
//在后面插入空白
bool CTonyBuffer::AddSpace2Tail(int nAddBytes)
{
    return SetSize(m_nDataLength+nAddBytes);
}
```

8.3.5.4 CutHead

`CutHead` 和 `InsertSpace2Head` 的功能正好相反，从缓冲区头删除一段空白。本函数破坏原数据被剪切部分。

```
//从前面剪切掉一段数据
void CTonyBuffer::CutHead(int nBytes)
{
    //防御性设计，如果给出的剪切空间大于原有缓冲区，则直接清空。
    if(m_nDataLength<=nBytes) SetSize(0);
    else
    {
        //这是从后向前 Move，因此，直接调用 memcpy 完成。
        memcpy(m_pData,m_pData+nBytes,m_nDataLength-nBytes);
        //大家请注意，这里笔者没有再 SetSize，由于内存池的原理，
        //ReMalloc 一个比较小的空间，一般都是直接返回原指针，
        //因此，此处也不多此一举了，直接就把空间长度修改为较小的长度即可
        m_nDataLength-=nBytes;
    }
}
```

8.3.5.5 CutTail

`CutTail` 是 `AddSpace2Tail` 的反义，也非常简单，本函数破坏原数据尾部被剪切部分。

```
//从后面剪切掉一段数据
void CTonyBuffer::CutTail(int nBytes)
{
    //防御性设计，剪切太多，直接清空
    if(m_nDataLength<=nBytes) SetSize(0);
    //同上，减小直接修改长度参数
    else m_nDataLength-=nBytes;
}
```

8.3.6 二进制拷贝函数

由于本类主要处理二进制缓冲区，因此，针对二进制格式的拷贝函数，可以说是最重要的函数功能。

8.3.6.1 BinCopyTo

这个函数的功能，是以本对象所保存的数据为来源，将内容拷贝到调用者指定的一个缓冲区。拷贝的数据多少，取决与调用者给出的缓冲区大小。本函数不破坏原数据。

```
//返回拷贝的字节数，拷贝失败，返回 0
int CTonyBuffer::BinCopyTo(
    char* szBuffer,                //调用者给出缓冲区指针
    int nBufferSize)               //调用者给出缓冲区大小
{
    //防御性设计
    if(!m_pData) return 0;         //如果内部无数据，返回 0
    if(!m_nDataLength) return 0;
    if(nBufferSize<m_nDataLength) return 0; //如果给定缓冲区小于本数据缓冲区，返回 0
    memcpy(szBuffer,m_pData,m_nDataLength); //执行拷贝动作
    return m_nDataLength;          //返回拷贝的字节长度
}
```

8.3.6.2 BinCopyFrom

本函数，将应用层给定的一个数据缓冲区的数据，拷贝到本对象中，由于本对象管理的是动态内存，因此，只要内存申请成功，拷贝一般都可以成功。本函数破坏原数据。

考虑到拷贝对象的多样性，做了一定重载，可以拷贝普通 C 语言的数据缓冲区，也可以拷贝另外一个同类对象的数据。

```
//拷贝一个二进制缓冲区的数据
int CTonyBuffer::BinCopyFrom(char* szData,int nDataLength)
{
    //防御性设计，如果给定的参数非法，清空本地数据，返回 0
    if((!szData)|| (0>=nDataLength))
    {
        SetSize(0);
        return 0;
    }
    if(!SetSize(nDataLength)) return 0; //重新设置
    memcpy(m_pData,szData,m_nDataLength);
    return m_nDataLength;
}

//拷贝同类，另外一个对象的数据
int CTonyBuffer::BinCopyFrom(CTonyBuffer* pBuffer)
{
    //这里调用上一函数完成功能，注意工具类的用法，直接访问目标内部数据区
    return BinCopyFrom(pBuffer->m_pData,pBuffer->m_nDataLength);
}
```

8.3.7 数值转换函数

很多时候，我们使用二进制网络协议，其信令往往是一个整数，或者一个字节，数值转换函数，就是帮助应用层，直接以网络字节序，把一个数字拷贝到缓冲区，或者反向获得出来。这在拼接二进制信令时非常有用。

8.3.7.1 网络字节序和本地字节序问题

在介绍数值转换函数前，有必要先给各位读者介绍一下网络字节序和本地字节序。

我们知道，PC 机，或者大多数嵌入式设备，32 位操作系统处理的整数，是 4Bytes，但内存单元仍然是以一个 Byte 作为最小单位。这就设计到一个排序问题。这个排序顺序，各家公司的设计原则是不一致的。

Intel 系列 CPU 执行的是低位在前，高位在后的原则，Windows 沿用了这一设计。但另一方面，Linux 由于继承了 Unix 的传统，遵循了 DEC 公司的设计方案，即高位在前，低位在后。我们看这个例子：

```
void TestByte(void)
{
    int n=0x11223344;
    int i=0;
    for(i=0;i<4;i++)
    {
        printf("%02x ",*((char*)&n+i));
    }
    printf("\n");
}
```

我们设定一个整数，为了便于观察，我们为其赋值十六进制的 11223344，然后我们按照字节顺序，从左至右打印看看（这段程序在 Windows 中实现）：

```
44 33 22 11 //这个顺序大家能看出什么吗？
```

我们可以看到，由于遵循 Intel 的低位在前高位在后的原则，整个数字在内存中的排列，与其在数字中的排列是相反的。这在单机中本来是没有问题的，但是，我们知道，商用数据传输工程的各个网络角色，很可能是跨平台开发，既通信的双方，不是一个操作系统，甚至某一方根本就不是 PC 机。

如果我们按照上述顺序，把该整数传递给一台其他计算机，比如一台古老的 DEC 工作站，则很可能被对方把这个整数理解为 0x44332211，这就完全错了。

因此，为了避免这类由于字节序问题导致的 bug，TCP/IP 协议栈约定，所有的二进制整数，或者短整数，在传递时，需要转化为网络字节序传递，接收方再逆向转回适合本系统的本地字节序。

这体现在函数里，就是下文中用到的 htonl，和 ntohs 等系统功能调用，前者把整数转化成网络字节序，后者逆向转回本地字节序。

当然，htons 和 ntohs 也是这个含义，仅仅是处理短整型（short）而已。

8.3.7.2 整数处理函数

整数处理函数，将一个整数（4Bytes）以网络字节序拷贝到缓冲区，或者以整数方式，求得本对象缓冲区最开始 4Bytes 的值。

```

//拷贝一个整数到本对象，执行网络字节序，破坏原有数据
bool CTonyBuffer::SetInt(int n)
{
    int nSave=htonl(n);          //以临时变量求得给定整数的网络字节序
    //拷贝到本地缓冲区，带 SetSize
    return BinCopyFrom((char*)&nSave,sizeof(int));
}
//以整数本地字节序求得缓冲区最开始 4Bytes 构成的整数的值，求值失败，返回 0
int CTonyBuffer::GetInt(void)
{
    //防御性设计，如果本对象没有存储数据，或者存储的数据不到一个整数位宽 4Bytes，返回 0
    if(!m_pData) return 0;
    if(sizeof(int)>m_nDataLength) return 0;
    return ntohl(*(int*)m_pData); //以头 4Bytes 数据，求得本地字节序返回
}

```

8.3.7.3 短整型处理函数

短整型处理函数与整型大同效益，大家可以比较看看。

```

bool CTonyBuffer::SetShort(short n)
{
    short sSave=htons(n);
    return BinCopyFrom((char*)&sSave,sizeof(short)); //拷贝的字节数变成短整型位宽
}
short CTonyBuffer::GetShort(void)
{
    if(!m_pData) return 0;
    if(sizeof(short)>m_nDataLength) return 0; //注意，此处的范围变成短整型的位宽
    return ntohs(*(short*)m_pData);
}

```

8.3.7.4 字节处理函数

字节处理函数则更为简单，由于一个 Byte 的位宽，连网络字节序的调整都不需要了。

```

bool CTonyBuffer::SetChar(char n)
{
    return BinCopyFrom(&n,sizeof(char)); //位宽 1Byte
}
char CTonyBuffer::GetChar(void)
{
    if(!m_pData) return 0;
    if(sizeof(char)>m_nDataLength) return 0; //位宽 1Byte
    return *(char*)m_pData;
}

```

8.3.8 二进制数据处理函数

这是前文二进制拷贝函数的补充，考虑到很多时候，我们需要拼接信令或报文，要求把新的数据增加到原有数据的前面或后面，而不破坏原有数据，因此定义这两个函数。

8.3.8.1 AddData

本函数将一段二进制数据添加到现有数据尾部，不破坏原有数据，这是最接近队列中 **AddLast** 的函数。

```
int CTonyBuffer::AddData(char* szData,int nDataLength)
{
    if((!m_pData)|| (0>=m_nDataLength))
    {    //防御性设计，如果原有数据为空，则直接执行拷贝动作
        return BinCopyFrom(szData,nDataLength);
    }
    int nOldSize=m_nDataLength;                //保留原有大小
    int nNewSize=m_nDataLength+nDataLength;    //求得新的大小
    if(!SetSize(nNewSize)) return 0;            //SetSize，其逻辑保留原有数据
                                              //注意，失败返回 0
    memcpy(m_pData+nOldSize,szData,nDataLength); //拷贝新数据到原有数据末尾
    return m_nDataLength;                       //返回新的大小
}
```

8.3.8.2 InsertData2Head

本函数将一段二进制数据，插入到原有数据头部，保留原有数据。

```
int CTonyBuffer::InsertData2Head(char* szData,int nDataLength)
{
    if((!m_pData)|| (0>=m_nDataLength))
    {    //防御性设计，如果原有数据为空，则直接执行拷贝动作
        return BinCopyFrom(szData,nDataLength);
    }
    //先在前插入足够空区，保证后续拷贝动作能成功。
    //根据 InsertSpace2Head 逻辑，原有数据可以获得保留
    if(!InsertSpace2Head(nDataLength)) return 0;
    //拷贝新数据到开始，注意拷贝的长度是新数据的长度，因此，不会破坏后续的原有数据
    memcpy(m_pData,szData,nDataLength);
    return m_nDataLength;                       //返回新的大小
}
```

8.3.9 文本字符串处理函数

很多时候，我们也会使用纯文本的网络协议，因此，笔者为 **Buffer** 类准备了两个文本字符串处理函数，方便应用层调用。

8.3.9.1 StrCopyFrom

这是以字符串方式拷贝一个字符串到本对象缓冲区。拷贝时，包括末尾的 `'\0'`。

```
int CTonyBuffer::StrCopyFrom(char* szString)
{
    int n=strlen(szString);                //先求出目标字符串的长度
    n++;                                   //长度+1，包括'\0'的位置
    return BinCopyFrom(szString,n);        //调用二进制拷贝函数完成动作
}
```

8.3.9.2 Printf

Printf 是很有趣的一个设计，笔者实际上是把 **SafePrintf** 的功能搬到了 **Buffer** 类中，很多时候，我们以一定协议规约设计文本型网络信令，如使用 **Json** 规范，都可能涉及到格式化打印的问题，因为这样比较方便，笔者在 **Buffer** 类中设计这个功能，主要目的就是为了方便应用层的使用。

```

//变参设计，返回打印的字符串总字节数(包含'\0'的位宽)
int CTonyBuffer::Printf(char* szFormat,...)
{
    //这一段在很多变参函数中都出现过，此处不再赘述
    char szBuf[TONY_BUFFER_STRING_MAX];
    int nListCount=0;
    va_list pArgList;
    va_start (pArgList,szFormat);
    nListCount+=Linux_Win_vsnprintf(szBuf+nListCount,
        TONY_BUFFER_STRING_MAX-nListCount,szFormat,pArgList);
    va_end(pArgList);
    if(nListCount>(TONY_BUFFER_STRING_MAX-1))
        nListCount=TONY_BUFFER_STRING_MAX-1;
    *(szBuf+nListCount)='\0';
    //最后调用 StrCopyFrom，将缓冲区内容拷贝到本对象
    return StrCopyFrom(szBuf);
}

```

这其中用到了一个宏，这是笔者工程库中较为通用的一个定义，即常规变参函数，处理的字符串总宽度，一般设置为 1024Bytes。

```
#define TONY_BUFFER_STRING_MAX (1024) //变参处理字符串的最大长度
```

8.3.10 数据比较函数

由于 Buffer 毕竟是数据的载体，因此很多时候，可能会面临一些数据比较，这种比较，又分为二进制比较和文本比较两类，笔者在类方法中予以实现，方便应用层操作。

```

//二进制比较
int CTonyBuffer::memcmp(char* szData,int nDataLength)
{
    //防御性设计
    if(!m_pData) return -1;
    if(!m_nDataLength) return -1;
    if(!szData) return -1;
    if(m_nDataLength<nDataLength) return -1;
    //使用 C 标准库的 memcmp 完成功能
    int nRet=::memcmp(m_pData,szData,nDataLength);
    return nRet;
}

//文本型比较
int CTonyBuffer::strcmp(char* szString)
{
    //防御性设计
    if(!m_pData) return -1;
    if(!m_nDataLength) return -1;
    if(!szString) return -1;
    //使用 C 标准库的 strcmp 完成功能
    int nRet=::strcmp(m_pData,szString);
    return nRet;
}

```

8.3.11 小结

经过上述介绍，相信各位读者应该对 Buffer 的设计思路，实现方式，接口设计都有了一个大概的了解，这里，笔者补充一点细节，请大家关注：

1、Buffer 类是一个典型的工具类，在放弃一定安全性的前提下，最大限度方便应用程序调用，因此，即使各个方法函数内部已经做了很多防御性设计，但还是很不够，要求调用者必须小心地使用本类对象，否则可能会出现 bug。

2、大家可以注意到，本类没有使用任何线程安全锁，原则上，工具类由于不对内部数据做任何私有屏蔽，做线程安全防护没有任何意义，本类的设计方向，仅对线程内部使用有效，应用层如果需要跨线程调用，请使用其他方法保证线程安全。

3、Buffer 类，同时兼容字符串和二进制两种操作模式，其中，很多方法互相之间是冲突的，这要求调用者必须很清楚当前的 Buffer 对象，存储的是哪类数据，使用中严禁变量转义，否则容易引起 bug。

8.4 静态 Buffer 类

完成了上述代码，我们手边已经有了一个可用的 Buffer 类了，看起来还很好用，一切似乎都已经 OK。

不过，真的 OK 了吗？

笔者在将上述动态 Buffer 类代入到工程中，以此为基础实现的一个 Relayer 中继服务器，公网上线就连续 n 个月工作，从来没有出过问题。看起来真的很 OK。一直到本书快写完的时候，出了一点小问题。

在最近一次工程的分析中，笔者发现了一个有趣的现象，内存池 90% 的动态分配行为，都来自于上述动态 Buffer 类，原因很简单，最主要的传输业务使用了该类，在传输收发过程中，都使用该对象做缓冲区，由于在变长信令的传输协议中，传输的信令的长度总是随时在变化的，因此，动态 Buffer 也总是不断地被 SetSize，于是，内部包含的内存块一次次地被重复分配。

笔者认为，不管怎么说，这么频繁地分配内存，对内存池的压力肯定很大，这，总不是一件好事情。

而另外一方面，几乎所有的网络信令设计，都有最大长度限制，除了流媒体传输，一般不太可能有无穷大的信令存在。事实上，连流媒体的传输，都是由很多细小的信令构成的，也不会不限制信令的长度。可以说，只要是使用了某种协议的传输链路，其内部的信令长度可能会变化，但最大长度是不会变的。站在数据传输的角度来说，我们设计一个协议，以及协议中的信令，往往第一步就已经规定了最大信令长度。

笔者很沮丧地发现，虽然前面笔者使用了大量的动态内存分配技巧，实现了动态内存 Buffer，但从笔者工程的实际应用上说，一个静态的 Buffer 类已经够用了，只要其内部的静态缓冲区大于最大信令长度即可。因为前级的函数已经保证了信令数据不会超过这一限制。

笔者思考了一下，抱着试试看的想法，按照前述动态 Buffer 类的 api 定义，重新设计了一个静态 Buffer 类，没想到效果异常地好，一经代入工程，Memory Pool 的压力不见了，程序运行也更为稳定。

因此，笔者在此处，同时提供动态和静态两种 buffer 类，供大家选用。各位读者也可以通过本文的示例，细心体会一下优化和 api 的关系。

8.4.1 类声明

这里请大家关注一个重点，Buffer 类的这次改版，是在上层应用工程已经成型的基础上改版，因此，必须 100%遵守动态 Buffer 类的 api 设计，否则就可能造成人为 bug，因此，静态 Buffer 类的声明与动态版本极其类似。

首先，静态 Buffer 类由于其内部是一个静态的常量，

```
#define TONY_SAFE_BUFFER_MAX_SIZE (132*1024) //暂定 132k，大家可以根据实际定义
```

由于我们坚持无缝替换，因此静态和动态 buffer 类的类公有方法可以说一模一样，不允许有任何变动。

```
class CTonyBuffer
{
public:
    //由于动静态 Buffer 类 api 完全一样，因此，公有函数构型必须保持一致。
    CTonyBuffer(CTonyBaseLibrary* pTonyBaseLib);
    CTonyBuffer(CTonyMemoryPoolWithLock* pMemPool);
    ~CTonyBuffer() {} //由于没有动态内存的释放任务，析构函数不做任何事
public:
    //////////////////////////////////////
    //二进制数据拷贝函数
    int BinCopyTo(char* szBuffer,int nBufferSize);
    int BinCopyFrom(char* szData,int nDataLength);
    int BinCopyFrom(CTonyBuffer* pBuffer);
    //////////////////////////////////////
    //文本数据拷贝构建函数
    int StrCopyFrom(char* szString);
    int Printf(char* szFormat,...);
    //////////////////////////////////////
    //尺寸设置函数
    //设置新的大小
    bool SetSize(int nSize);
    //在前面插入空白
    bool InsertSpace2Head(int nAddBytes);
    //在后面插入空白
    bool AddSpace2Tail(int nAddBytes);
    //从前面剪切掉一段数据
    void CutHead(int nBytes);
    //从后面剪切掉一段数据
    void CutTail(int nBytes);
    //////////////////////////////////////
    //数值转换函数
    bool SetInt(int n);
    int GetInt(void);
    bool SetShort(short n);
    short GetShort(void);
    bool SetChar(char n);
    char GetChar(void);
```

```

////////////////////////////////////
//二进制数据追加函数
//追加数据到最后，返回新的数据长度
int AddData(char* szData,int nDataLength);
//插入数据到最前面，返回新的数据长度
int InsertData2Head(char* szData,int nDataLength);
////////////////////////////////////
//数据比较函数
int memcmp(char* szData,int nDataLength);
int strcmp(char* szString);
////////////////////////////////////
//服务函数
bool IHaveData(void);
public:
    char m_pData[TONY_SAFE_BUFFER_MAX_SIZE];    //请注意这里，m_pData 变为静态数组
    int m_nDataLength;
    CTonyMemoryPoolWithLock* m_pMemPool;        //保留 MemPool，是为了 Debug 方便
};

```

8.4.2 构造函数和析构函数

静态 buffer 类的构造函数，为了维护 api 的统一，保持和动态类相同的构型。原则上，Memory Pool 内存池对象的指针已经不再需要，不过，考虑到 Debug 方便，我们需要从内存池对象内部获得 Debug 对象的指针，因此予以保留。

这里模糊了一个概念，m_nDataLength 在动态 buffer 类中有两个含义，第一是数据长度，第二是具体使用内存块的大小，在静态类中，由于内存块为栈空间自动分配的数组，因此其大小不再有意义，因此，m_nDataLength 在本类中，仅表示有效数据的长度。同时，m_pData 也不允许再为 null。

```

CTonyBuffer::CTonyBuffer(CTonyBaseLibrary* pTnyBaseLib)
{
    m_pMemPool=pTnyBaseLib->m_pMemPool;        //保留内存池对象
    m_nDataLength=0;                            //数据长度为 0
}
CTonyBuffer::CTonyBuffer(CTonyMemoryPoolWithLock* pMemPool)
{
    m_pMemPool=pMemPool;                        //保留内存池对象
    m_nDataLength=0;                            //数据长度为 0
}
bool CTonyBuffer::IHaveData(void)              //已有数据标志
{
    //由于使用静态数组，m_pData 永远有意义，因此，此处仅判断 m_nDataLength 的值
    if(0>=m_nDataLength) return false;
    return true;
}

```

析构函数由于不再需要释放内存，因此不做什么事，在类声明中直接实现空函数。

8.4.3 缓冲区设置函数

缓冲区设置函数是变化比较大的一组函数，请大家仔细观察与动态 **buffer** 类的区别。

8.4.3.1 SetSize

SetSize 是变化最大的一个函数，由于取消了动态内存申请，因此这个函数实际上没有意义。不过为了保持 **api** 的一致性，必须予以保留。不过，**SetSize** 的另外一个功能仍然有用，即判断新设置的数据缓冲区是否超界。

```
bool CTonyBuffer::SetSize(int nSize)
{
    if(TONY_SAFE_BUFFER_MAX_SIZE<nSize)
    {
        //如果超界，报警，返回假，即告知调用者分配失败
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::SetSize(): ERROR! nSize=%d\n",nSize);
        return false;
    }
    m_nDataLength=nSize;    //数据长度置为设置尺寸，
                           //这其实还是把 m_nDataLength 作为缓冲区大小提示
    return true;
}
```

8.4.3.2 InsertSpace2Head

在缓冲区已有数据前面插入空白区段，由于没有动态内存分配，这个动作编程了一个单纯的缓冲区已有数据向后移动的行为。


```

bool CTonyBuffer::InsertSpace2Head(int nAddBytes)
{
    if (0>=m_nDataLength)
    {
        //如果没有原始数据，则视为设置缓冲区大小
        m_nDataLength=nAddBytes;
        return true;
    }
    //这个计算相对复杂，首先缓冲区最大尺寸恒定，因此，它与原有数据的差值，是新插入空区的
    //最大可能值，因此，此处必须做判断
    if (nAddBytes>(TONY_SAFE_BUFFER_MAX_SIZE-m_nDataLength))
    {
        //条件不满足，则报警返回
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::InsertSpace2Head(): ERROR!
            nAddBytes=%d, m_nDataLength=%d, too big!\n",
            nAddBytes,m_nDataLength);
        return false;
    }
    //注意此处的大括号，是限定变量 szBuffer 的作用范围
    char szBuffer[TONY_SAFE_BUFFER_MAX_SIZE];
    memset(szBuffer,'\0', TONY_SAFE_BUFFER_MAX_SIZE);
    //第一次，把原有数据拷贝到新缓冲区，已经偏移了 nAddBytes 的位置
    memcpy(szBuffer+nAddBytes,m_pData,m_nDataLength);
    m_nDataLength+=nAddBytes;
    //第二次，将新缓冲区有效数据拷贝回本对象缓冲区
    memcpy(m_pData,szBuffer,m_nDataLength);
    //之所以这么复杂的拷贝，主要就是为了规避前文所述的“从前拷贝”和“从后拷贝”问题
}
return true;
}

```

8.4.3.3 AddSpace2Tail

```

bool CTonyBuffer::AddSpace2Tail(int nAddBytes) //在后面插入空白
{
    if (0>=m_nDataLength)
    {
        //如果没有原始数据，则视为设置缓冲区大小
        m_nDataLength=nAddBytes;
        return true;
    }
    //判断新设置的尺寸大小是否合适
    if (nAddBytes>(TONY_SAFE_BUFFER_MAX_SIZE-m_nDataLength))
    {
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::AddSpace2Tail(): ERROR!
            nAddBytes=%d, m_nDataLength=%d, too big!\n",
            nAddBytes,m_nDataLength);
        return false;
    }
    m_nDataLength+=nAddBytes; //后面追加比较简单，修改 m_nDataLength 的值即可
    return true;
}

```

8.4.3.4 CutHead

```
void CTonyBuffer::CutHead(int nBytes) //从前面剪切掉一段数据
{
    if(0>=m_nDataLength)
    {
        //没有原始数据，剪切无意义，报警，宣告失败
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::CutHead(): ERROR! m_nDataLength=%d, too small!\n",
            m_nDataLength);
        m_nDataLength=0;
        return;
    }
    if(nBytes>m_nDataLength)
    {
        //如果剪切的数据长度大于原始数据长度，报警
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::CutHead(): ERROR! m_nDataLength=%d, nBytes=%d,
            too small!\n",
            m_nDataLength,nBytes);
        m_nDataLength=0;
        return;
    }
    m_nDataLength-=nBytes; //求出新的数据长度
    //后面数据向前拷贝，“挤出”原有数据
    memcpy(m_pData,m_pData+nBytes,m_nDataLength);
    return;
}
```

8.4.3.5 CutTail

```
void CTonyBuffer::CutTail(int nBytes) //从后面剪切掉一段数据
{
    if(0>=m_nDataLength)
    {
        //没有原始数据，剪切无意义，报警，宣告失败
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::CutTail(): ERROR! m_nDataLength=%d, too small!\n",
            m_nDataLength);
        m_nDataLength=0;
        return;
    }
    if(nBytes>m_nDataLength)
    {
        //如果剪切的数据长度大于原始数据长度，报警
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::CutTail(): ERROR! m_nDataLength=%d, nBytes=%d,
            too small!\n",
            m_nDataLength,nBytes);
        m_nDataLength=0;
        return;
    }
    m_nDataLength-=nBytes; //缩短长度就是剪切尾部
    return;
}
```

8.4.4 二进制拷贝函数

二进制拷贝函数与动态 `buffer` 类很类似。

```
int CTonyBuffer::BinCopyTo(char* szBuffer,int nBufferSize)
{
    //防御性设计，条件不满足则报警，返回 0，表示没有拷贝成功
    if(m_nDataLength>nBufferSize)
    {
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::BinCopyTo(): ERROR! nBufferSize=%d,m_nDataLength=%d\n",
            nBufferSize,m_nDataLength);
        return 0;
    }
    if(!szBuffer)
    {
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::BinCopyTo(): ERROR! szBuffer=null\n");
        return 0;
    }
    //执行真实的拷贝逻辑
    memcpy(szBuffer,m_pData,m_nDataLength);
    return m_nDataLength;    //返回拷贝的字节数
}
```

```

int CTonyBuffer::BinCopyFrom(char* szData,int nDataLength)
{
    //防御性设计
    if (TONY_SAFE_BUFFER_MAX_SIZE<nDataLength)
    {
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::BinCopyFrom(): ERROR! nDataLength=%d, too big!\n",
            nDataLength);
        return 0;
    }
    if(!szData)
    {
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::BinCopyTo(): ERROR! szData=null\n");
        return 0;
    }
    if(0>=nDataLength)
    {
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::BinCopyTo(): ERROR! 0>=nDataLength\n");
        return 0;
    }
    //真实的拷贝动作
    memcpy(m_pData,szData,nDataLength);
    m_nDataLength=nDataLength;
    return m_nDataLength;        //返回拷贝的字节数
}

int CTonyBuffer::BinCopyFrom(CTonyBuffer* pBuffer)
{
    //拷贝另外一个 buffer
    return BinCopyFrom(pBuffer->m_pData,pBuffer->m_nDataLength);
}

```

8.4.5 数值转换函数

```

bool CTonyBuffer::SetInt(int n)           //设置一个整数，网络格式
{
    int nSave=htonl(n);
    return BinCopyFrom((char*)&nSave,sizeof(int));
}

int CTonyBuffer::GetInt(void)             //获得一个整数，返回本地格式
{
    if(0>=m_nDataLength) return 0;
    int* pData=(int*)m_pData;
    int nRet=*pData;
    return ntohl(nRet);
}

```

```

bool CTonyBuffer::SetShort(short n)      //设置一个短整数，网络格式
{
    short sSave=htons(n);
    return BinCopyFrom((char*)&sSave,sizeof(short));
}
short CTonyBuffer::GetShort(void)        //获得一个短整数，返回本地格式
{
    if(0>=m_nDataLength) return 0;
    short* pData=(short*)m_pData;
    short sRet=*pData;
    return ntohs(sRet);
}
bool CTonyBuffer::SetChar(char n)        //设置一个字节
{
    *m_pData=n;
    m_nDataLength=sizeof(char);
    return true;
}
char CTonyBuffer::GetChar(void)          //得到 m_pData 第一个字节的值
{
    return *m_pData;
}

```

8.4.6 二进制数据处理函数

二进制数据处理函数，变化也很小，动静态 **buffer** 类的主要差异，在基础的一些函数。业务层变化不大。

8.4.6.1 AddData

```

//二进制数据追加函数
//追加数据到最后，返回新的数据长度
int CTonyBuffer::AddData(char* szData,int nDataLength)
{
    int nNewSize=m_nDataLength+nDataLength;      //求得新的尺寸
    if(TONY_SAFE_BUFFER_MAX_SIZE<nNewSize)        //防御性判断
    {
        m_pMemPool->m_pDebug->Debug2File(
            "CTonyBuffer::AddData(): ERROR! m_nDataLength=%d,
            nDataLength=%d, too big!\n",
            m_nDataLength,nDataLength);
        return 0;
    }
    //做真实的拷贝动作
    memcpy(m_pData+m_nDataLength,szData,nDataLength);
    m_nDataLength=nNewSize;
    return m_nDataLength;
}

```

8.4.6.2 InsertData2Head

```

//插入数据到最前面，返回新的数据长度
int CTonyBuffer::InsertData2Head(char* szData,int nDataLength)
{
    if(!InsertSpace2Head(nDataLength))           //先试图插入空白到最前
    {
        m_pMemPool->m_pDebug->Debug2File("CTonyBuffer::InsertData2Head():
ERROR! m_nDataLength=%d, nDataLength=%d, too big!\n",
        m_nDataLength,nDataLength);
        return 0;
    }
    memcpy(m_pData,szData,nDataLength);          //成功则拷贝
    return m_nDataLength;
}

```

8.4.7 文本字符串处理函数

```

int CTonyBuffer::StrCopyFrom(char* szString)      //拷贝一个字符串到内部
{
    int nDataLength=strlen(szString)+1;          //求出字符串数据长度，
                                                    //+1 表示包含'\0'
    return BinCopyFrom(szString,nDataLength);    //调用 BinCopyFrom 完成拷贝
}
int CTonyBuffer::Printf(char* szFormat,...)       //变参打印构造一个字符串
{
    char szBuf[TONY_SAFE_BUFFER_MAX_SIZE];       //注意，最大长度为静态 buffer
                                                    //的最大长度

    int nListCount=0;
    va_list pArgList;
    va_start (pArgList,szFormat);
    nListCount+=Linux_Win_vsnprintf(szBuf+nListCount,
        TONY_SAFE_BUFFER_MAX_SIZE-nListCount,szFormat,pArgList);
    va_end(pArgList);
    if(nListCount>(TONY_SAFE_BUFFER_MAX_SIZE-1))
        nListCount=TONY_SAFE_BUFFER_MAX_SIZE-1;
    *(szBuf+nListCount)='\0';
    //以上为变参处理段落，已经多处出现，此处不再赘述
    return StrCopyFrom(szBuf);                   //调用 StrCopyFrom 完成拷贝
}

```

8.4.8 数据比较函数

```
int CTonyBuffer::memcmp(char* szData,int nDataLength) //二进制比较
{
    //防御性设计
    if(0>=m_nDataLength) return -1;
    if(!szData) return -1;
    if(m_nDataLength!=nDataLength) return -1;
    int nRet=::memcmp(m_pData,szData,nDataLength); //调用系统函数完成
    return nRet;
}

int CTonyBuffer::strcmp(char* szString) //字符串比较
{
    //防御性设计
    if(0>=m_nDataLength) return -1;
    if(!szString) return -1;
    int nRet=::strcmp(m_pData,szString); //调用系统函数完成
    return nRet;
}
```

8.4.9 小结

本小节中，笔者在动态 buffer 类的基础上，又给大家提供了一段相同 api 的，功能也几乎完全一致的静态 buffer 类，这种看似矛盾的设计，其实体现着很深层的含义，在这里，笔者认为需要和大家沟通一些细节。

1、在商用工程开发中，很难找到真正放之四海皆真理的“完美程序”，通常的说法是，只有更好，没有最好。一段代码，在某处工作很好，换个地方，换个程序员，也许工作就不尽人意，这是很正常的现象，希望大家正确理解。

2、有的时候，理论不能代表一切。从理论上，动态 buffer 无意具有很大优越性，高效，精炼，但是，它对内存池的冲击很大，对于商用系统有潜在的危害，因此，笔者虽然写出了动态 buffer 类，但近期的一些开发确多用静态类来实现，这没有什么道理，完全是实践的感觉。程序设计是一门实践性的科学，建议各位读者在以后的开发中，不要迷信理论，不要人云亦云，一切设计，建议具体问题具体分析，以自己的判断为准。

3、软件设计是一种平衡，任何获利都是有代价的，动态 buffer，节约了内存，获得了效率，但是，带来了内存池冲击，同时带来了频繁的锁操作（内存池有锁），有时候反而降低了性能。静态 buffer 类，看似浪费了一点内存，但是，带来了系统的稳定性，这里面没有对错，只有取舍，而取舍之道，在于大家对实际业务流程的理解和基本计算原理的掌握。

4、模块化是程序开发的基础，而实现模块化开发，其关键就是 api，大家可以注意到，在 api 相同的情况下，笔者上层程序将感觉不到两个 buffer 类的切换。这无疑是非常方便的设计，任何时候，我们都可以任意替换某个模块，而不会影响到其他模块，这就是 api 最大的用途。

5、但是，请注意，api 相同是一个很难做到的任务，不但函数名，函数构型相同，连内部的成员变量的含义、返回值的默认含义也要相同，这实际上是需要仔细分析的。比如 m_nDataLength 这个值的含义，在动静态 buffer 类中就不尽一样，笔者为了达到 api 相同的目的，设计 SetSize 这个函数的时候，很费了一番脑筋，因此，这类模块级替代设计，需要对 api 有很深的理解才行。请各位读者以后谨慎操作。

8.5 PopBuffer

我们已经介绍过，PopBuffer 是最基本的队列方式，一般就基于一块内存缓冲区工作，对外提供队列式操作方法，方便应用程序操作。工作时，PopBuffer 更多的类似于一种粘合类的功能，即依托一块应用层传进来的缓冲区工作。

8.5.1 PopBuffer 基本需求分析

PopBuffer 缓冲区主要用于数据传输中，信令打包工作。前文说过，由于确定传输所需要成本过高，服务器每秒钟能进行的绝对收发次数，是有一定限制的。这就造成了服务器吞吐量的低下。

为了解决这个问题，在既保证安全的前提下，进一步提高传输速率，就势必要把信令打包发送。

我们知道，一般的中继转发服务器，在收到一个信令后，往往是放入一个异步队列，然后接受线程立即回来，准备接收下一笔数据。由于接收和发送的不同步，每次发送线程准备好，发送下一笔数据时，往往会在任务队列中看到多条淤塞的信令，常见的做法，就是利用 PopBuffer，将淤塞信令打成一个整包，发送出去，接收端再进行解包。

因此，PopBuffer 通常的需求，就是从一个队列中，弹出最前面的几十笔数据，进行发送，或者将收到的一个信令包，粘合进来，按照队列的方式，以 GetAndDeleteFirst 方式拆解出来，逐一解析执行。所以说，PopBuffer 队列，是信令打包和拆包最方便的工具，应用非常广泛。

8.5.2 基本数据结构介绍

PopBuffer 类，由于具有粘合类的特性，其缓冲区通常是外部应用层准备好的报文收发 buffer，是线性编址的二进制缓冲区，同时，其主要功能用于打包和拆包，换言之，PopBuffer 与队列管理相关的业务变量，必须内置于队列缓冲区内，与队列数据一起被收发，才能被接收方正确解析。这是一种很特殊的类设计。

因此，我们首先就要有一个结构体来定义 PopBuffer 最核心的一些业务变量，这个变量结构体，将放置于队列缓冲区头部，与后续数据一起被收发。

在队列和很多多元素类设计中，笔者习惯使用 Token（元素）来称呼内部的数据单位。请各位读者注意定名。

```
//PopBuffer 队列管理变量结构体
typedef struct _TONY_POP_BUFFER_HEAD_
{
    int m_nTokenCount;           //内部包含的元素个数
    int m_nAllBytesCount;        //使用的总字节数
}STonyPopBufferHead;           //定义的结构体变量类型
//习惯性写法，定义了结构体，立即定义其长度常量
const ULONG STonyPopBufferHeadSize=sizeof(STonyPopBufferHead);
//由于 PopBuffer 缓冲区的线性连续编址特性，这个队列头结构体被设计成位于缓冲区最开始的地方
//因此，队列第一个元素真实的开始点，是缓冲区开始处向后偏移 STonyPopBufferHeadSize 长度
//这个宏定义出队列元素数据区开始指针（字符型指针）
#define TONY_POP_BUFFER_TOKEN_DATA_BEGIN(p) \
    (((char*)p)+STonyPopBufferTokenHeadSize)
```

同时，我们针对每一个 Token，需要有一个小结构体，来描述这个 Token 的细节。这是该 Token 的 Head 结构体。


```

//每个 Token 的头结构体
typedef struct _TONY_POP_BUFFER_TOKEN_HEAD_
{
    int m_nDataLength;          //标示该 Token 的数据长度
}STonyPopBufferTokenHead;      //定义的结构体变量类型
//结构体长度常量
const ULONG STonyPopBufferTokenHeadSize=sizeof(STonyPopBufferTokenHead);
//如果一笔准备推入队列的数据长度为 n，则该 Token 的总长度可以用该宏计算
#define TONY_POP_BUFFER_TOKEN_LENGTH(n) (n+STonyPopBufferTokenHeadSize)
//如果已知一个 Token 的起始指针为 p，则该 Token 的数据区开始处可以用该宏计算
#define TONY_POP_BUFFER_FIRST_TOKEN_BEGIN(p) \
    (((char*)p)+STonyPopBufferHeadSize)

```

图 8.3: PopBuffer 缓冲区内部数据示意图

我们可以用一个简要的图，简要了解一下 PopBuffer 内部的数据区分配情况。

这里给大家提示一点函数型宏的写法，大家可能注意到，在函数型宏的计算中，凡是涉及到指针计算的，笔者往往会将指针转化成为 char* 型指针来处理，大家可以看看这个定义中的 ((char*)p) 就是强制指针类型转换。

```

#define TONY_POP_BUFFER_FIRST_TOKEN_BEGIN(p) \
    (((char*)p)+STonyPopBufferHeadSize)

```

这是一个习惯，我们知道，所有的指针，由于其基础类型不同，其+1 时的步距未必相同，我们做个简单的实验看看：

```

void TestPoint(void)
{
    char* pChar=0;          //定义一个 char*
    int* pInt=0;            //定义一个 int*
    int i=0;
    for(i=0;i<5;i++)
    {
        printf("%p ",pChar++); //以+1 为步距，检测 char* 的变化
    }
    printf("\n");
    for(i=0;i<5;i++)
    {
        printf("%p ",pInt++);  //以+1 为步距，检测 int* 的变化
    }
    printf("\n");
}
//这是输出结果
00000000 00000001 00000002 00000003 00000004//char*+1，真的只增加 1Byte
00000000 00000004 00000008 0000000C 00000010//int*+1，增加 4Byte，就是 int 的位宽

```

大家可以看到，随着指针指向的数据类型位宽的变化，指针+1 的步距也在发生变化。由于我们的函数型宏，并没有什么强制类型判断，原则上，任何指针都可以传递进宏进行计算，因此，必须预防应用层传递进来错误类型的指针，导致计算不准。

通常情况下，我们使用的二进制缓冲区，在 C 和 C++语言中，都是字符串型的数组，因为这样最好计算，+1 的步距就是 1，不会引发歧义。

因此，笔者的习惯，凡是函数型宏中，涉及到指针的计算，一般都是强制转化成 `char*` 后计算，这样，需要累加的数字，非常简单，加几，指针就向后 **Move** 几，不会因为计算错误导致崩溃。

当然，这也有个前提，笔者所有工程使用的 **buffer**，都是最基本的字符型数组，需要进行特定的指针转换时，临时使用强制指针类型转换使用，但平时的表述一定是字符型的，以配合上述习惯。

提示：如果各位读者喜欢使用函数型宏的话，建议在所有的指针计算中，做上述的转换设计，避免潜在的 bug。

8.5.3 基本类模型

```
//基本的 PopBuffer 类，本类同时兼顾粘合类和独立类两种特性
class CTonyPopBuffer
{
public:
    //注意，这是粘合类的特有构造函数，内部缓冲区是外部传入，
    CTonyPopBuffer(char* szBuffer,          //缓冲区指针
                    int nBufferSize,        //缓冲区尺寸
                    bool bInitFlag=true);    //是否初始化标志
    ~CTonyPopBuffer();

public:
    //实现“后粘合”的具体函数，即实现粘合的方法
    void Set(char* szBuffer,int nBufferSize);
    //清空整个数据区，仅仅是数据归零，缓冲区不释放
    void Clean(void);
    //内部信息打印函数，Debug 用，相当于前文的 PrintfInfo
    void PrintInside(void);
    //能否正确工作的标志函数
    bool ICanWork(void);

public:
    //队列最经典的功能，追加到末尾，此处兼容普通缓冲区和 Buffer 类
    int AddLast(char* szData,int nDataLength);
    int AddLast(CTonyBuffer* pBuffer);

public:
    //获得当前内部元素个数
    int GetTokenCount(void);
    //获得当前使用的所有字节数（包含管理字节）
    int GetAllBytes(void);
    //根据即将推送进队列的数据长度，判断内部剩余空间是否够用
    bool ICanSave(int nDataLength);

public:
    //获得第一个元素的长度
    int GetFirstTokenLength(void);
    //获取第一个元素，这是普通 buffer 版本，需要应用层保证缓冲区长度足够
    //用 GetFirstTokenLength 可以查询第一个元素的长度
    int GetFirst(char* szBuffer,int nBufferSize);
    //获得第一个元素，这是 Buffer 类版本
    int GetFirst(CTonyBuffer* pBuffer);
    //删除第一个元素
    bool DeleteFirst(void);
    //提取并删除第一元素，就是从队列中弹出第一个元素
    int GetAndDeleteFirst(char* szBuffer,int nBufferSize);
    int GetAndDeleteFirst(CTonyBuffer* pBuffer);
```

```

public:
    //枚举遍历所有数据，提交回调函数处理，并且删除数据，返回经过处理的 Token 个数
    int MoveAllData(_TONY_ENUM_DATA_CALLBACK pCallBack,PVOID pCallParam);
public:
    char* m_pBuffer;          //最关键的内部缓冲区指针
    int m_nBufferSize;        //内部缓冲区长度
private:
    //这是队列头的指针，注意，这个指针并不是动态申请的内存块，而是指向缓冲区 m_pBuffer 头
    STonyPopBufferHead* m_pHead;
};

```

8.5.4 构造函数和析构函数

PopBuffer 由于是纯正的粘合类，因此其构造函数和析构函数比较特殊，最重要的是，析构函数不做任何事，因为没有什么数据需要摧毁。

```

CTonyPopBuffer::CTonyPopBuffer(char* szBuffer,
    int nBufferSize,
    bool bInitFlag)
{
    m_pHead=null;          //初始化各种管件变量
    m_pBuffer=null;
    m_nBufferSize=0;
    Set(szBuffer,nBufferSize);    //调用后粘合的 Set 函数，实现粘合
    if(bInitFlag) Clean();        //如果需要初始化，则清空整个队列
}
CTonyPopBuffer::~CTonyPopBuffer(){}

```

这里面特别值得一提的是 **bInitFlag** 这个变量。我们知道，PopBuffer 是粘合类，换言之，内部缓冲区不是它自己申请的，而是应用层传入的。

通常情况下，C 语言在远堆动态申请一块内存，这种内存由于是各个程序不断重用的，一般显得很“脏”，即里面的数据是乱的随机数，或者是根据其他程序逻辑规律化的一些数字，如果直接使用，这就是典型的“野变量”，会导致不可知的结果，甚至引发程序崩溃。因此，不管是变量，还是动态申请的内存块，在使用前，一般需要按照本程序的逻辑做一定的初始化，方能放心使用，这也是大家常见的“变量需要赋初值原理”。

不过，我们仔细分析，可能发现，PopBuffer 往往会面临两种情况：

1、在信令发送方，一般是应用层准备好一块内存，传进来准备进行信令打包，此时的缓冲区，确实需要初始化，否则会出错。

2、在信令的接收方，收到远端传来的一个信令包，该包已经由发送方的 PopBuffer 按照规定初始化好，并且构造好队列，此时如果初始化，就会丢失信息，引发错误，反而有害。

有鉴于此，笔者在 PopBuffer 的构造函数中，设定了一个 **bInitFlag** 变量，又应用层根据实际情况，决定传入的缓冲区是否进行初始化。

8.5.5 工具服务函数

工具服务函数，主要指与业务数据无关，属于 PopBuffer 自己的服务功能函数。

8.5.5.1 ICanWork

这个函数前面我们已经做了很多介绍，主要就是为所有的内外使用者，提供一个明确的判断，检测本对象内部各个关键变量是否已经正确赋值，是否可以正常工作。

```
bool CTonyPopBuffer::ICanWork(void)
{
    //依次检测所有关键变量
    if(!m_pBuffer) return false;
    if(!m_nBufferSize) return false;
    if(!m_pHead) return false;
    return true;
}
```

8.5.5.2 Set

这是真实的粘合动作，就是把外部传入的缓冲区，与内部缓冲区指针挂接，准备开始工作。

```
void CTonyPopBuffer::Set(char* szBuffer,int nBufferSize)
{
    m_pBuffer=szBuffer;                //挂接缓冲区
    m_nBufferSize=nBufferSize;
    m_pHead=(STonyPopBufferHead*)m_pBuffer;    //定义队列头指针
}
```

这里特别值得一提的，是最后一句，给 `m_pHead` 赋值。

我们知道，队列最关键的两个业务变量，一个就是元素个数，一个就是总字节数，我们在上面，使用了一个结构体来封装，这就是队列头结构体 `STonyPopBufferHead`。

同时，我们也约定，这个队列头，一定位于缓冲区最开始的地方，因此，`m_pBuffer` 最开始之处，总是这个队列头。

不过，由于 `m_pBuffer` 本身是一个字符串指针，也就是二进制指针，其指针类型并不是这个队列头结构体，后文使用起来很不方便，需要不断地进行强制指针类型转换。

因此，笔者在类中设置了一个成员变量 `m_pHead`，从最初的粘合方法 `Set` 开始，就做好这种强制指针类型转换，方便后文调用，同时也减少多次重复动作可能引发的 `bug`。

提示：这是一个很好的习惯，当我们被迫以 `char*` 类型存储一根指针，但实际使用中，又不断需要强制转换成其他类型指针使用时，可以考虑再设置一根同名异类的指针，保存该指针数值，减少后续的强制指针类型转换动作，简化程序，减少 `bug`。

8.5.5.3 Clean

`Clean` 可以说是真正的初始化函数，它将从逻辑上，将队列置为一个空队列，可以插入数据，这个清空，不是指缓冲区释放，而是队列逻辑上，定义无任何元素在内。

```
void CTonyPopBuffer::Clean(void)
{
    if(m_pHead)                //注意，此处已经开始使用 m_pHead
    {
        m_pHead->m_nTokenCount=0;    //所有 Token 总数置为 0
        //总消耗的字节数，置为队列头的长度。
        //这表示，PopBuffer 最后输出的总字节数，是包含这个队列头长度的。
        //同时也表示，即使 PopBuffer 内部一个 Token 都没有，其字节数也不为 0
        m_pHead->m_nAllBytesCount=STonyPopBufferHeadSize;
    }
}
```

8.5.5.4 PrintInside

前文大家可能已经看出笔者的习惯，在很多核心的，底层的模块中，由于调试不便，笔者习惯于内置打印方法，将内部数据以某种格式打印出来，以便在需要的时候，进行跟踪和观察。这个 `PrintInside` 就是典型的 `Debug` 函数。

注意，这个函数最难的一点，是不破坏原有数据的前提下，遍历输出全队列的内容，这意味着，`GetAndDeleteFirst` 的功能不能使用，只能自行构建遍历循环。

```
//格式化输出内部数据
void CTonyPopBuffer::PrintInside(void)
{
    if(!ICanWork()) //防御性设计
        TONY_XIAO_PRINTF("CTonyPopBuffer::PrintInside(): \
            ERROR! m_pBuffer=null\n");
    return;
}
//定义元素区开始的指针
char* pTokenBegin=TONY_POP_BUFFER_FIRST_TOKEN_BEGIN(m_pBuffer);
//定义元素的头指针
STonyPopBufferTokenHead* pTokenHead=
    (STonyPopBufferTokenHead*)pTokenBegin;
//利用 pTokenHead，偏移计算本 Token 的数据开始点
//请注意，这里传入的 pTokenHead，就不是字符串型，如果前文函数型宏定义中
//没有做强制指针类型转换，此处已经出错。
char* pTokenData=TONY_POP_BUFFER_TOKEN_DATA_BEGIN(pTokenBegin);
int i=0;
//预打印整个队列的信息，即元素个数，字节数。
TONY_XIAO_PRINTF("CTonyPopBuffer::PrintInside(): Token: %d Bytes: %d\n",
    m_pHead->m_nTokenCount,m_pHead->m_nAllBytesCount);
for(i=0;i<m_pHead->m_nTokenCount;i++)
{
    //注意，由于队列中存储的数据，可能是文本型，也可能是二进制型
    //笔者在此准备了两条打印语句，使用时，根据情况选择
    //TONY_XIAO_PRINTF("[%d] - %s\n", //格式化输出文本
        pTokenHead->m_nDataLength,pTokenData);
    //dbg_bin(pTokenData,pTokenHead->m_nDataLength); //格式化输出二进制
    //开始迭代计算，根据本 Token 长度，计算下一 Token 起始点
    pTokenBegin+=
        TONY_POP_BUFFER_TOKEN_LENGTH(pTokenHead->m_nDataLength);
    //修订相关 Token 头等参变量
    pTokenHead=(STonyPopBufferTokenHead*)pTokenBegin;
    pTokenData=TONY_POP_BUFFER_TOKEN_DATA_BEGIN(pTokenBegin);
}
}
```

8.5.6 业务查询函数

由于队列业务需要不断和应用层交互，应用层很多时候需要查询队列内部存储情况，因此，笔者在 `PopBuffer` 内部设置了一些业务查询函数。

8.5.6.1 GetTokenCount

这是非常常用的一个函数，就是获得队列内部的元素个数。

```
int CTonyPopBuffer::GetTokenCount(void)
{
    if(!ICanWork()) return 0;           //防御性设计
    return m_pHead->m_nTokenCount;      //返回元素个数
}
```

该函数虽然简单，但在各种队列中，通常是使用率最频繁的函数，经常我们需要弹空队列中所有的元素时，会使用如下的调用模型：

```
void Func(CTonyPopBuffer* pPopBuffer)
{
    while(pPopBuffer->GetTokenCount())    //只要元素个数不为 0，就是有元素
    {
        pPopBuffer->GetAndDeleteFirst(...); //弹出第一个元素，开始做事
        //...
    }
}
```

8.5.6.2 GetAllBytes

其实作为队列对象，在使用中，大家很少关心其具体占用的字节数，不过，由于 PopBuffer 主要用于信令打包，以便发送，内部数据的字节数，就变得非常重要，通常被用作信令报文的发送字节数。本函数就是提供该值。

```
int CTonyPopBuffer::GetAllBytes(void)
{
    if(!ICanWork()) return 0;           //防御性设计
    return m_pHead->m_nAllBytesCount;    //返回所有占用的字节数
}
```

8.5.6.3 ICanSave

由于 PopBuffer 主要工作于外部应用层提供的一个有限缓冲区，该缓冲区随着不断的 AddLast，会被逐渐“撑满”，由于我们的队列，是适应各种长度的数据，理论上，小于缓冲区最大长度的数据，都可以被放入队列，因此，剩余空间的检查，就变得非常重要，本函数就是实现这种检查功能。

//检查剩余空间是否够存储给定的数据长度

```
bool CTonyPopBuffer::ICanSave(int nDataLength)
{
    int nLeaveBytes=0;                    //准备变量，求得剩余空间
    if(!ICanWork()) return false;       //防御性设计
    //剩余空间=缓冲区总长度-AllBytes，我们知道，AllBytes 里面已经包含了队列头长度
    //因此，这种计算是正确的。
    nLeaveBytes=m_nBufferSize-m_pHead->m_nAllBytesCount;
    //判断语句，注意，进入的长度，需要利用函数型宏进行修饰。
    //由于每个 Token 有一个小的头，这个头的长度，需要叠加进来判断，否则就不准确
    if(TONY_POP_BUFFER_TOKEN_LENGTH(nDataLength) > (ULONG)nLeaveBytes)
        return false;
    else
        return true;
}
```

8.5.7 添加 AddLast

AddLast 可以说是队列最重要的函数之一，所有的数据插入，由此完成。在笔者的开发规约中，队列的 AddLast 在添加成功时，返回添加的数据字节数，否则返回 0。

```
//针对普通缓冲区的 AddLast
int CTonyPopBuffer::AddLast(char* szData,int nDataLength)
{
    int nRet=0;                                //准备返回值
    //防御性设计
    if(!szData) goto CTonyPopBuffer_AddLast_End_Process;
    if(0>=nDataLength) goto CTonyPopBuffer_AddLast_End_Process;
    if(!ICanWork()) goto CTonyPopBuffer_AddLast_End_Process;
    //如果剩余空间不够添加，则跳转返回 0
    if(!ICanSave(nDataLength)) goto CTonyPopBuffer_AddLast_End_Process;
    { //请注意，这个大括号不是 if 语句开始，上文 if 语句如果成立，已经 goto 跳转
      //此处主要是为了限定变量的作用域，gcc 中，不允许在 goto 语句之后声明变量
      //此处的大括号，是为了重新开辟一个堆栈区，以便声明局部变量
      char* pTokenBegin=                        //利用 AllBytes 求出队列最尾的偏移
          m_pBuffer+m_pHead->m_nAllBytesCount;
      STonyPopBufferTokenHead* pTokenHead=      //强制指针类型转换为 Token 头指针
          (STonyPopBufferTokenHead*)pTokenBegin;
      char* pTokenData=                        //求出 Token 数据区的指针
          TONY_POP_BUFFER_TOKEN_DATA_BEGIN(pTokenBegin);
      //请注意具体的添加动作
      pTokenHead->m_nDataLength=nDataLength; //先给元素头中的长度赋值
      memcpy(pTokenData,szData,nDataLength); //memcpy 数据内容到缓冲区
      m_pHead->m_nTokenCount++;                //元素个数+1
      //请注意，这里 AllBytes 添加的是包括元素头的所有长度，而不仅仅是数据长度
      m_pHead->m_nAllBytesCount+=              //AllBytes 加上增加的长度
          TONY_POP_BUFFER_TOKEN_LENGTH(nDataLength);
      nRet=nDataLength;                        //但返回值纯数据长度
    }
    CTonyPopBuffer_AddLast_End_Process:        //结束跳转点
        return nRet;                           //返回结果
}
//针对 Buffer 类的 AddLast
int CTonyPopBuffer::AddLast(CTonyBuffer* pBuffer)
{
    if(!pBuffer) return 0;                    防御性设计
    //调用上一函数，实现功能
    return AddLast(pBuffer->m_pData,pBuffer->m_nDataLength);
}
```

这里特别请各位读者关注一个细节，虽然我们队列内部管理时，每个 Token 的实际长度，是 Token 头+数据长度，但是，这个是队列内部的细节，不需要，也不应该应用程序了解，因此，AddLast 实际表示成功的返回值，还是添加的数据长度。

这里面有两点考虑：首先，应用程序很可能使用如下的构型完成添加动作，并判断成功与否：


```

bool Func(CTonyPopBuffer* pPopBuffer)
{
    char szData[100];           //准备一个缓冲区
    int n=SafePrintf(szData,100,"xxx"); //格式化一段字符串信令（随便打的，没意义）
    n++;                         //长度+1，包括最后的'\0'
    if (n==pPopBuffer->AddLast(szData,n)) //判断添加到队列是否成功
        return true;
    else
        return false;
}

```

大家注意，如果我们的 AddLast 返回的是 Token 的长度，而不是数据的长度，应用层的判断就会出现问題，导致 bug。

当然，各位读者可以说，我们在调用者 Func 中修饰一下 n，把 n 加上 Token 头的长度来判断可不可以，从这个例子看，当然可以，但这带出了第二个问题。

PopBuffer 的 Token 头的定义，是一个纯粹的內部定义，在以后的工作中，随着需求的变化，很可能我们需要维护这个头，在里面添加一些其他内容，我们以结构体来封装，目的就是在将来保留这种增减信息量的方便性。

但如果我们的 AddLast 把这个头长度暴露给使用者，使用者必须使用这个长度来修饰判断语句，则这个数据就不再私有。

我们知道，一个数据一旦不是私有，修改起来就非常麻烦，万一哪天我们需要在 Token 头中添加字段，增加长度，我们将被迫遍历所有使用 PopBuffer 的工程代码，检查每一处 AddLast，看这种增加是否会引发新的 bug，这将带来极大的工作量和 bug 隐患。

因此，这里有一个很重要的原则：“C++面向对象的类设计，除了有意识地将一些成员变量和成员函数以私有方式管理，避免无意义的访问外，还应该对自己的一些内部常量，私有约定，私有数据结构做一定的屏蔽，避免引发不必要的 bug。”

8.5.8 提取 GetAndDeleteFirst

从队列中提取数据，也是队列很重要的基本功能。本小节即着重向大家介绍这个函数族。

8.5.8.1 关于“等停”的针对性设计

可能各位读者有点费解，队列原则上是先进先出，只要提取数据，一定就是从队列头弹出数据，即数据取出，就应该删除，为什么笔者还需要很费力地设计出 GetFirst, DeleteFirst, GetAndDeleteFirst 等等一系列函数。

这里有个很重要的原因，就是前文所述的，在商用数据传输工程中的“等停”确定传输功能。我们来看一段样例代码：

```

bool SendJob(CTonyPopBuffer* pPopBuffer)
{
    CTonyBuffer RWBuffer;           //准备一个 Buffer 缓冲区
    pPopBuffer->GetFirst(&RWBuffer); //从队列中取出第一条
    if (!send(&RWBuffer,...)) return false; //如果发送失败，返回假，等待下一轮重试
    pPopBuffer->DeleteFirst();        //成功则删除第一条
    return true;                     //返回真
}

```

这是在“等停”式传输中常见的一种逻辑，由于我们从队列中取出第一条数据，并不确定本次传输是否会成功，因此，还不敢从队列中删除第一条。一直等到传输成功后，才会删除。

由于网络数据传输行为通常都是围绕一个任务队列在循环，因此，某一次传输失败，只要数据不丢失，并不算很大问题，通常的处理就是传输程序先返回假，等待下一轮重试。这类错误并不算 bug。

因此，为了适应这类需求，队列除了提供传统的 `GetAndDeleteFirst` 之外，还需要提供 `GetFirst`，`DeleteFirst` 来做特殊适应。下面我们逐一介绍。

8.5.8.2 数据的安全操作模型

在笔者设计队列时，曾经有一个讨论，就是提取时，应用层应该以何种方式提取。在 C 语言中，一个上层程序希望从下层功能层获取数据，通常有下面几种做法：

1、上层准备好缓冲区，通过参数将缓冲区指针传递给下层函数，下层函数将数据填充到该缓冲区中，然后返回长度，上层程序由此获得数据。

2、下层程序直接根据数据长度，动态申请合适大小的内存块作为缓冲区，将数据填充其中，然后将数据区指针返回给上层程序，上层程序使用后，释放该指针。

3、下层函数提供回调函数接口，上层准备好回调函数，并将回调函数指针在调用中传递给下层函数，下层函数，也是动态准备好缓冲区，填充数据，通过回调函数将数据传送给上层，当回调返回时，下层函数释放缓冲区后，再返回。

说实话，C 语言并不是一种很方便的编程语言，这里就体现出来了。由于缺乏类似于其他高级语言的变长数组类型，程序员必须自行处理数据传递的细节，导致了上述问题的出现。

上述方法，第二种，不符合“谁申请，谁释放”原则，有一定风险，一般不建议使用。第三种，符合“谁申请，谁释放”原则，但调用太繁琐，不方便，也不推荐使用。算来算去，唯一能用的，就是第一种。

由于第一种，上层函数准备好缓冲区，必须先知道多大的缓冲区才能满足使用，因此，笔者才在队列类中增加了一个查询函数，`GetFirstTokenLength`，帮助上层程序准备缓冲区。

8.5.8.3 `GetFirstTokenLength`

这个函数是获得队列中第一个元素的数据长度，好帮助应用层准备缓冲区，开始接收第一个元素的数据。

```
int CTonyPopBuffer::GetFirstTokenLength(void)
{
    if(!ICanWork()) return 0;                //防御性设计
    char* pFirstTokenBegin=                   //利用宏计算第一个 Token 起始点
        TONY_POP_BUFFER_FIRST_TOKEN_BEGIN(m_pBuffer);
    STonyPopBufferTokenHead* pFirstTokenHead= //获得 Token 头指针
        (STonyPopBufferTokenHead*)pFirstTokenBegin;
    return pFirstTokenHead->m_nDataLength;    //返回头中包含的数据长度
}
```

8.5.8.4 `GetFirst`

本函数返回第一个 Token 的数据，有两个重载类型。

```

//以普通缓冲区方式获得第一个 Token 数据，上层程序保证缓冲区足够大，并传入供检查
int CTonyPopBuffer::GetFirst(char* szBuffer,int nBufferSize)
{
    int nRet=0; //准备返回参数
    //防御性设计
    if(!ICanWork())
        goto CTonyPopBuffer_GetFirst_End_Process;
    //判定队列是否为空
    if(!GetTokenCount())
        goto CTonyPopBuffer_GetFirst_End_Process;
    //判断给定的参数区是否合法
    if(GetFirstTokenLength()>nBufferSize)
        goto CTonyPopBuffer_GetFirst_End_Process;
    { //注意，这个大括号不是 if 语句的大括号，是限定变量作用域
        char* pFirstTokenBegin= //寻找第一个 Token 起始点
            TONY_POP_BUFFER_FIRST_TOKEN_BEGIN(m_pBuffer);
        STonyPopBufferTokenHead* pFirstTokenHead= //获得 Token 头指针
            (STonyPopBufferTokenHead*)pFirstTokenBegin;
        char* pFirstTokenData= //获得 Token 数据指针
            TONY_POP_BUFFER_TOKEN_DATA_BEGIN(pFirstTokenBegin);
        memcpy(szBuffer,pFirstTokenData, //拷贝数据到缓冲区
            pFirstTokenHead->m_nDataLength);
        nRet=pFirstTokenHead->m_nDataLength; //返回值设定
    }
CTonyPopBuffer_GetFirst_End_Process:
    return nRet;
}
//以 Buffer 类方式获得第一个 Token 数据，本函数破坏 Buffer 原有内容
//由于 Buffer 类本身是动态内存管理，因此，不存在缓冲区问题
int CTonyPopBuffer::GetFirst(CTonyBuffer* pBuffer)
{
    //防御性设计，
    if(!ICanWork()) return 0;
    if(!pBuffer->SetSize(GetFirstTokenLength())) return 0;
    if(!pBuffer->m_nDataLength) return 0;
    //调用上一函数，实现真实的 GetFirst 功能。
    return GetFirst(pBuffer->m_pData,pBuffer->m_nDataLength);
}

```

8.5.8.5 DeleteFirst

本函数删除第一个 Token。本函数实现比较困难的是需要关注几个细节：

- 1、需计算第一个 Token 起始点，以及其内部各指针。
- 2、需根据第一个 Token 的起始点，极其长度，计算第二个 Token 的起始点。
- 3、从第二个 Token，到队列末尾的字节长度，由于是挤出式操作，需要利用 `memcpy`，把第二个 Token 之后的数据，向前 Move 到第一个 Token 处。
- 4、由于是从后向前 Move，因此，`memcpy` 可以直接使用，不需要考虑从前拷贝还是从后拷贝问题。


```

//Buffer 类方式，弹出第一个元素数据
int CTonyPopBuffer::GetAndDeleteFirst(CTonyBuffer* pBuffer)
{
    if(!ICanWork()) return 0; //防御性设计
    int nRet=GetFirst(pBuffer); //获得第一个元素数据
    DeleteFirst(); //删除第一个元素
    return nRet;
}

```

8.5.9 MoveAllData

这是纯粹为了方便使用设置的一个接口函数，其中心思想就是实现把一个队列弹空的逻辑，中间使用回调函数来处理每笔 Token 数据。首先，我们来看看回调函数构型

```

//数据枚举回调函数,返回真，继续枚举，直到结束，否则直接结束循环
typedef bool (*_TONY_ENUM_DATA_CALLBACK)(char* szData, //数据指针
                                           int nDataLength, //数据长度
                                           void* pCallParam); //代传的参数指针
//这是笔者一个习惯，写出一个回调函数构型后，立即写个 Demo，后续使用者可以直接拷贝使用
//static bool EnumDataCallback(char* szData,int nDataLength,void* pCallParam);

```

这里大家可以看出笔者写回调函数的一个通用设计方案：

1、凡是遍历等带循环效果的回调函数，构型返回值，一般为 bool，一个默认的约定是，回调函数返回真，遍历循环继续，返回假，则循环中断。

2、除了基本的业务参数外，一定要帮助上层回调函数设计者，代传一根 void* 的参数指针，这在很多时候会起很大作用。

3、写出回调函数构型后，一般立即写一个 static 的函数声明，以注释方式保存，使用者可以直接拷贝这个声明道类声明中使用。

函数原型如下：

```

//枚举遍历所有数据，提交回调函数处理，并且删除数据，返回经过处理的 Token 个数
//为了避免过多的动态内存分配细节，或者缓冲区的组织，本函数没有使用 GetFirst
int CTonyPopBuffer::MoveAllData(
    _TONY_ENUM_DATA_CALLBACK pCallBack,    //回调函数指针
    PVOID pCallParam)                    //代传的 void*参数指针
{
    int nMovedTokenCount=0;                //统计变量返回值
    bool bCallbackRet=true;                //这是记录回调函数返回值的变量
    if(!pCallBack)                         //防御性设计，如果回调函数为空，不做事
        goto CTonyPopBuffer_MoveAllData_End_Process;
    if(!ICanWork())                        //如果条件不满足，不做事
        goto CTonyPopBuffer_MoveAllData_End_Process;
    while(m_pHead->m_nTokenCount)           //以 TokenCount 为 while 循环的条件
    {
        char* pFirstTokenBegin=            //求得第一个 Token 开始点
            TONY_POP_BUFFER_FIRST_TOKEN_BEGIN(m_pBuffer);
        STonyPopBufferTokenHead* pFirstTokenHead= //求得第一个 Token 的头指针
            (STonyPopBufferTokenHead*)pFirstTokenBegin;
        char* pFirstTokenData=              //求得第一个 Token 的数据指针
            TONY_POP_BUFFER_TOKEN_DATA_BEGIN(pFirstTokenBegin);
        //请注意此处回调，将第一个 Token 数据传给回调函数，且获得回调函数返回值
        bCallbackRet=pCallBack(pFirstTokenData,
            pFirstTokenHead->m_nDataLength,
            pCallParam);
        DeleteFirst();                      //删除第一个 Token
        nMovedTokenCount++;                  //返回值+1
        if(!bCallbackRet) break;            //如果回调返回假，中止循环
    }
CTonyPopBuffer_MoveAllData_End_Process:
    return nMovedTokenCount;                //返回处理的 Token 个数
}

```

8.5.10 PopBuffer 小结

在本小节中，笔者向各位读者展示了一个基于静态 Buffer 的粘合工具类 PopBuffer 队列的实现方法，这里，笔者希望读者关注几点细节：

8.5.10.1 关于缓冲区

在很多时候，笔者会提出 C 语言普通缓冲区这个概念，这是相对于 Buffer 类对象而言的。在 C 语言中，我们知道，二进制缓冲区就是线性编址的内存，可以存储任何数据，反过来说，任何数据，均可以表现为(char* szData,int nDataLength)这种形式，这也是笔者在工程实战中常用的数据表示方法。

学习过 COM 的读者可能理解，在远程调用过程中，最困难的莫过于传递各种类型的参数，COM 接口为此，无所不用其极，甚至创造了智能指针这种变量来专门处理。但这些都带来了使用者学习的困难性和使用上的不便。

在笔者设计商用数据传输工程时，由于网络上各个角色往往不都是 Windows 系统，因此根本无法使用 COM 或 DCOM 接口来完成远程跨平台过程调用，必须自己完成。

为了简化设计，笔者就给自己和项目团队约定，**所有的数据，在外部传输过程中，或者存储过程中，尽量表现为无属性的纯正二进制数据段，即上文的(char* szData,int**

nDataLength)形式，当某段代码需要以业务方式使用数据时，请自行根据实际业务做强制指针类型转换。

这个设计思想，大大简化了商用数据传输工程的设计难度，也使得跨平台，复杂数据类型的传输成为可能。因此，这个重点思想，请大家关注。

8.5.10.2 以需求为引导的 api 设计

大家在本类的设计中，可能发现笔者的接口设计有点随意化，很多时候，一个接口函数，仅仅是针对一个常见需求来设计，这其实是体现了工程化设计思想。

在工程设计中，并不强调一个工具类设计出来，要适应所有的可能需求情况，因为这实际上是不可能的。很多工程库模块，其接口函数都是不断维护和完善的过程，当我们遇到一个需求，需要一个新的接口，我们就及时添加该接口，但一般不会主动去假设，以后可能有什么接口。根据经验，这类假设，一般都是无用功。

这在以后的模块设计中，大家还能多次看到这种设计思路。

8.5.10.3 api 的初步

当然，作为一个标准的队列类，一些必要的功能接口是必须的，比如 `AddLast`，`GetAndDeleteFirst` 这几个接口，以笔者的理解，这些可以算作队列类统一的 api。

在笔者任何一个队列模块的设计中，这些接口是必须的，并且，其参数，其返回值的含义，永远不变，永远就是那么几个意思。

这是因为，一个 api 一旦固定下来，就成为上层模块和下层模块的连接标准，在符合 api 规约的前提下，我们随时可以更换上层的应用层，也可以重新定义下层的功能层的实现定义，这些都不影响程序的运行。

这就是 api 最关键的功能，以一定的规范，约束交互的双方，使双方在某种程度上，割裂一些必然的联系，最终达到双方可以随意更换。

事实上，笔者的 `PopBuffer`，脱离其服务的应用层，单独改版了很多次，但由于接口从未变过，因此，每次修改，上层应用层无需做任何更改，大大方便了改版，减少了很多 bug。

笔者希望各位读者学习这一思想，在模块开发中引入 api 的概念，可以有效帮助大家提升开发效率。

8.5.10.4 函数型宏的使用

在上述设计中，笔者使用了大量的函数型宏，大家可以发现，本小节介绍的几个函数型宏，主要起到了固化计算的作用。

我们知道，在 C 和 C++ 语言中，都有常量的设计，为的是固化某些数字，收拢访问接口，方便以后修改。

笔者个人理解，函数型宏和常量一样，唯一不同的是，它固化的的是一个计算关系，也是为了收拢访问接口，方便以后修改。

这种固化思维，请各位读者关注，很多时候，这是减少 bug，降低改版工作量的关键。

当然，C++ 推荐的 `inline` 函数，也能达到类似的目的，不过，这仅仅是笔者个人的习惯，各位读者可以自行选择自己喜欢的固化方式。

8.5.10.5 每段逻辑只写一次

除了函数型宏，各位读者可能还能注意到，笔者在开发中的一个重要习惯，即每个逻辑仅写一次。

如 `AddLast`，`GetAndDeleteFirst` 这几个接口的不同参数重载，基于 `Buffer` 类的函数基本上都是调用普通缓冲区类的函数完成。

这看似一个细节，但请大家关注，这是一个很重要的开发习惯。

在开发中，我们很多时候需要对一个调用，一个逻辑进行修改，以适应新的需求，如果我们在多处实现同一逻辑，在修改时我们也势必需要修改多处，这会带来很多的工作量和 bug 风险。

每个逻辑只写一次，其实和函数型宏一样，是一种收拢访问的固化写法，为的是使同一逻辑，只有一个实现点，降低修改的工作量，降低 bug。

同时，这样的设计，还有两个好处，一是使程序变得很简洁，二是强迫程序员不断抽象，这些都是保证软件产品质量的关键技巧。

因此，建议各位读者细心体会这些细节背后，所体现的深刻思想。

8.6 MemQueue

PopBuffer 虽然已经实现了队列常见的功能，能够支持一般的调用，但细心的读者可能会发现，它有一个最大的缺陷，就是效率偏低。原因很简单，PopBuffer 为了实现粘合类的作用，并针对其主要业务----信令打包和拆包做了特定的限制，其工作空间是一段线性编址的连续二进制缓冲区。

因此，PopBuffer 的 GetAndDeleteFirst 动作的效率并不高。最大的 loading 出现在，弹出第一个 Token 之后，后续的 Token 有一个显式的向前 Move 的动作。这在高频插入和弹出的场合，显然效率非常低。显然，PopBuffer 并不适应高性能服务器、任务机等场合。

为了解决这个问题，笔者专门又开发了 MemQueue 这个模块。该模块内部全部采用动态内存分配，队列采用链表方式管理，并采取特定优化，取消遍历循环，实现了高速场合下的任务队列模型。

8.6.1 动态队列的管理原则

在笔者设计动态队列 MemQueue 的时候，笔者主要针对 PopBuffer 的不足，提出了几点原则，在设计中坚持遵守。

1、链表的设计，每个子节点的设计，不允许使用对象，原因很简单，对象不能被内存池管理，无法实现内存重用，会导致内存碎片。而动态队列作为未来服务器的高速核心任务处理机，势必会出现高频的进出操作，导致内存频繁被分配和释放，内存碎片可以说必然发生，因此，只能用内存池管理的动态内存块实现。

2、采用资源锁方式，并以功能类和锁封装类两个类方式构建。这是由于队列内部各个方法之间耦合严重，彼此经常互相调用，很多时候，公有方法需要调用公有方法，把锁做进功能类，很可能导致不必要的麻烦，甚至不可实现，因此，按照资源锁概念，分两层封装。

3、考虑到这个队列很可能被用作主要的信令流缓存，而信令一般不允许丢失，MemQueue 必须支持磁盘 Dump 的方法，即在必要时，可以把内部数据全部存入磁盘文件做永续保存，下次启动自动读入。

4、考虑到信令打包需求，队列不仅仅支持弹出一个 Token，也应该支持对 PopBuffer 的批量弹出需求，反之，数据压入队列时，也应该同时支持这两个数据对象。

5、由于商用数据传输工程中，一个服务器内部可能有多个队列，因此，队列的所有 Debug 和日志输出，必须能显式打印自己的身份，否则程序员无法观察某条数据具体存在于哪个队列。

6、队列的优化方向：空间换时间，以时间作为第一优化目标。队列中不允许出现大规模的递归，避免遍历时发生崩溃。

7、MemQueue 由于常用于多线程环境下，因此必须做到线程安全。

8.6.2 基本数据结构介绍以及优化考虑

动态队列的基本数据结构，主要包括队列管理的相关变量设计，链表子节点的数据结构设计，以及优化相关设计。

8.6.2.1 基本数据结构

笔者在设计 MemQueue 时，首先考虑了整体的管理模型，我们知道，如果要提升效率，势必需要废弃 PopBuffer 中连续内存编址的问题，因为它无法保证挤出第一个 Token 时的内存 Move 效率。

最好的解决方案，当然是以纯指针方式管理的链表，一个 Token 的插入和删除，都仅仅是修改某个指针的值，再申请或释放一块内存。再加上内存池的优化帮助，效率非常高。

因此，笔者在设计 MemQueue 的数据结构时，主要考虑的就是如何构建链表对象，我们知道，一个链表，关键是其每个节点的数据结构设计，如果设计得不好，其效率可能会很低。经过考虑和试验，笔者设计了如下的数据结构：

```
//MemQueue 链表节点数据结构
typedef struct _TONY_XIAO_QUEUE_TOKEN_HEAD_
{
    int m_nDataLength;           //存储的业务数据长度
    char* m_pBuffer;             //指向业务数据块的指针
    struct _TONY_XIAO_QUEUE_TOKEN_HEAD_* m_pNext; //指向下一节点的指针
}STonyXiaoQueueTokenHead;       //定义的新的结构体变量类型
//笔者习惯，写完结构体，立即声明其长度常量，方便后续的内存申请。
const ULONG STonyXiaoQueueTokenHeadSize=sizeof(STonyXiaoQueueTokenHead);
```

其次，我们知道，MemQueue 要想管理好这个链表，必须内置链表的首节点，即至少应该以上述结构体，定义一个首节点指针 m_pHead。

当然，作为队列的标准服务，也应该提供队列中元素个数变量统计 m_nTokenCount，由于 MemQueue 没有 PopBuffer 那样，需要将整个队列所有数据打包收发的需求，因此，上节中的 AllBytes 无需再统计。

8.6.2.2 优化考虑

一般说来，我们设计程序，都应该遵循“先实现，后优化”的顺序，笔者对 MemQueue 的优化，也是最后完成的，本书中，仅仅是为了章节需要，方便大家理解，先在此做介绍，请大家注意，这并不是说，笔者赞同，程序一开始就进行优化。

在笔者设计 MemQueue 的过程中，对效率做了很多测试和思考。其中，笔者最主要关注的效率就是如何规避遍历循环。这是因为，链表的基本特点是，从头尾部分入链和出链很快，但是，遍历检索效率很低。

笔者仔细分析了队列的特点，GetAndDeleteFirst，不用考虑，这是从头部弹出，效率很高。但 AddLast，如果以常规的链表方式管理，则每次 AddLast 都需要遍历到队列最末尾添加，这个效率堪忧。

因此，经过思考，笔者为 MemQueue 设计了一个加速因子，m_pLast，这是一个非关键变量，即当其为空时，AddLast 会按照常规流程，遍历整个链表后，将数据添加到末尾，如图 8.4。




图 8.4: 传统的链表遍历方式

不过, 在本类中, `AddLast` 会多做一件事, 就是每次找到链表最尾, 完成添加动作后, 会将最末尾一个节点的指针, 填充到加速因子 `m_pLast`。这样, 当下次 `AddLast` 时, 可以直接检索到这个 `m_pLast`, 一步到位, 将数据插入到最后, 大幅度提升效率。如图 8.5。




图 8.5: 使用加速因子的插入模式

大家可以推论一下, 我们仅仅添加了一个指针变量加速因子 `m_pLast`, 就立即将一个队列的 `AddLast` 操作由 $O(n)$ 提升到 $O(1)$, 并且还不影响原有的功能。

这也是笔者经常自己设计底层库的原因, 很多时候, 我们可以在工程中使用一些开源的, 或者商用的工程库, 这当然是好事, 可以大幅度节约我们的成本, 提升程序员的生产力。

不过, 万事有利就有弊, 使用别人的库, 非常简单, 但也对我们屏蔽了很多细节, 使我们通常希望针对我们的业务, 做一点立竿见影的针对性优化, 几乎变得不可能。

以本小节为例, 我们知道, 任何一个通用库中的链表类, 一般不太可能针对队列的需求, 设置这个 `m_pLast` 的加速因子, 原因很简单, 这个加速因子仅仅针对队列的 `AddLast` 起作用, 不具备通用性, 如果程序员用这个类来做一个堆栈, 则根本无用。

大家可以想一想, 如果我们使用通用的链表类来完成 `MemQueue`, 这个效率是不是一定不如我们自己设计的这个管理模型。

提示: 笔者在这里建议, 对于通用库的使用, 可以用, 但要有所节制, 不能人云亦云, 什么都是别人的库好, 必要时, 针对自己面临的工程项目的某些关键点, 自己实现一下底层, 做一点针对性的优化, 可以起到事半功倍的作用。

8.6.3 基本功能类声明

首先, 我们来看看 `MemQueue` 的私有变量区

```
//动态内存队列类
class CTonyXiaoMemoryQueue
{
private:
    //由于 MemQueue 使用动态内存分配，理论上是没有限制的
    //但我们知道，程序中任何数据结构，都应该有边界，否则可能会造成 bug
    //这里使用一个内部变量，强行界定边界值，为队列最大长度做个上限
    int m_nMaxToken;                //最大的 Token 限制
    int m_nTokenCount;              //队列中有效 Token 个数

    STonyXiaoQueueTokenHead* m_pHead;    //队列头指针
    STonyXiaoQueueTokenHead* m_pLast;    //加速因子，队列尾指针
    CTonyLowDebug* m_pDebug;             //debug 对象指针
    CTonyMemoryPoolWithLock* m_pMemPool; //内存池指针（本类依赖内存池）
    char m_szAppName[TONY_APPLICATION_NAME_SIZE];

```

其次，我们看看构造函数和析构函数构型

```
public:
    CTonyXiaoMemoryQueue(CTonyLowDebug* pDebug, //debug 对象指针
        CTonyMemoryPoolWithLock* pMemPool,     //内存池指针
        char* szAppName,                        //应用名，这里代队列名
        int nMaxToken=TONY_CHAIN_TOKEN_MAX);    //最大 Token 上限
    ~CTonyXiaoMemoryQueue();

```

此处，需要特别说明的是构造函数中的 **AppName** 应用名这个参数。我们知道，在一个完整的商用数据传输服务器中，队列几乎无处不在，下行队列，上行队列，中继转发队列，很多时候，甚至针对每个客户，会有一个队列，随着服务客户数的增多，这类队列可能同时存在成千上万个实例。

而很多时候，程序员出于 **Debug** 的考虑，或者出于性能的观察需求，需要检查一个信令的生死，利用队列内部的一些输出函数，将队列的内容打印出来观察。如果队列没有在功能上做定名，所有的队列输出信息混杂到一起，程序员就很难看懂这类流水账似的天书。

笔者有鉴于此，在设计 **MemQueue** 时，特别强调要定义一个“应用名（**AppName**）”，以此为队列命名，标定本队列是服务于哪个应用的队列，如客户端上传信令队列等，在内部输出时，一并输出，帮助程序员做信息定位。

同时，构造函数中也确定了队列的最大长度 **nMaxToken**，这里使用了一个宏默认值，其定义如下：

```
#define TONY_CHAIN_TOKEN_MAX 1024
```

即一般情况下，如果不做特殊约定，队列默认最大 1024 个 Token，超出此限，**AddLast** 会失败，由于队列在工作中，总是不断进出数据，因此，这个 1024 实际上约定的是最大并发数据量，这个已经能满足大多数应用场合了，当然，如果有特殊需要，可以考虑更换这个数字。

下面，我们看看常用的工具函数

```
public:
    bool ICanWork(void);                //这个太熟悉了吧，是否可以工作标志
    void CleanAll(void);                //清除所有 Token，队列清空
    int GetFirstLength(void);           //获得第一个 Token 数据长度
                                         //功能和目的同 PopBuffer
    int GetTokenCount(void){return m_nTokenCount;} //获得所有 Token 总数，内联
    void PrintInside(void);             //遍历打印所有队列内部 Token 数据

```

这里我们介绍一下队列常用功能，基本等同 **PopBuffer**，此处不再赘述。

8.6.4 构造函数和析构函数

```
//构造函数，初始化所有变量
CTonyXiaoMemoryQueue::CTonyXiaoMemoryQueue(
    CTonyLowDebug* pDebug,
    CTonyMemoryPoolWithLock* pMemPool,
    char* szAppName,
    int nMaxToken)
{
    m_nMaxToken=nMaxToken;           //保留最大 Token 上线
    m_pLast=null;                    //注意，此处加速因子 m_pLast 被设置为 null
    m_pDebug=pDebug;                 //保留 debug 对象指针
    m_pMemPool=pMemPool;              //保留内存池指针
    SafeStrcpy(m_szAppName,szAppName, //保留 AppName
        TONY_APPLICATION_NAME_SIZE);
    m_pHead=null;                    //注意，m_pHead 被设置为 null，表示无 Token
    m_nTokenCount=0;                 //Token 计数器被设置为 0
}
//析构函数，清除所有的 Token，释放申请的内存
CTonyXiaoMemoryQueue::~CTonyXiaoMemoryQueue()
{
    if(m_pHead)
    {
        DeleteATokenAndHisNext(m_pHead); //此处递归释放
        m_pHead=null;
    }
}
```

这里需要特别说明一点的是析构函数，大家可能发现，析构函数中使用了递归，按照前文所述 C 和 C++ 无错化程序设计方法，在大规模链表中应该尽量避免递归。这似乎和前述原则相违背。

笔者是这么考虑的，**MemQueue** 主要用于高性能服务器的任务机，报文、信令的中转转发，主要用于缓冲收发不同步导致的报文淤塞情况。由于在服务器正常运行时，报文总是有进有出，因此，**MemQueue** 就其使用模式来说，内部数据主要体现的是并发数据量，准确地讲，是并发后，由于服务器短期内处理不过来，造成的淤塞数据量。

这类淤塞在商用数据传输系统中，往往会由于某个节点的网络不通顺，某个时间段的业务峰值而加大，但随着时间的流逝，业务压力逐渐趋于正常，淤塞又会逐渐消除。因此，总的来说，**MemQueue** 一般不会在其内部长期淤塞大量数据，其总长度一般说来有限。

笔者在构造函数中设置队列默认最大 Token 数为 1024 个，就是这个道理，事实上，从 **MemQueue** 设计出来以后，在正常的服务器业务压力下，这个上限从未被打破过。因此，由于递归导致可能的系统崩溃，一般说来不太可能，笔者此处就使用了递归开发，以简化程序。

另外，我们换一个角度，也可以理解这种设计。

我们知道，析构函数一般说来是应用服务器程序总的退出时才会运行，此时，即使不释放内存，由于进程退出，操作系统拆除进程空间，内部数据自然消亡，因此，退出时的内存泄漏没有太大问题。

我们所要关心的，无非是队列中信令的保护问题。服务器运行期间的信令流，一般说来分为两类，一类是记录运行时各个客户端，服务器，网络节点的运行状态，一类是数据库操作相关信令。

对于前者，状态通告信令，是随着本次运行期间而有效的，当服务器结束运行，这类状态一般已经失去其意义，无需保存。因此，如果是这类信令，退出时原则上是抛弃。

而对于后者，信令需要做长期保护，断线保护，因此，**MemQueue** 为应用层提供了磁盘文件存储函数，目的就是应用层实现数据保护。原则上，当系统退出进行到队列退出时，应用层的前级逻辑，一定已经调用了队列的磁盘数据存储函数，将数据保存到磁盘上了，因此，此时内部的数据，也已经失效，无需再进行保护。

因此，本类的析构函数，对数据的处理较为粗糙，没有做更多的保护策略。目的也是为了降低开发成本。

8.6.5 辅助工具函数

MemQueue 的辅助工具类函数，主要应对内部的一些防御性设计，**Debug** 设计，以及基础的一些查询函数等。

8.6.5.1 ICanWork

ICanWork 主要用于判定 **Debug** 对象和内存池对象指针是否完备，本对象具有基本的操作条件。

```
bool CTonyXiaoMemoryQueue::ICanWork(void)
{
    if(!m_pDebug) return false;           //检查 debug 对象指针
    if(!m_pMemPool) return false;         //检查内存池指针
    return true;
}
```

8.6.5.2 CleanAll

CleanAll 是提供队列对象运行过程中的清空，内部调用 **DeleteFirst** 循环完成，规避了递归，但是，由于 **DeleteFirst** 有额外的队列维护动作，因此效率不是最高，析构函数没有使用本函数，就是希望直接实现能提高删除效率。

```
void CTonyXiaoMemoryQueue::CleanAll(void)
{
    if(!ICanWork()) return;               //防御性设计
    while(DeleteFirst()) {}                //循环删除第一个对象，直到队列为空
                                           //注意，本类的 DeleteFirst 返回删除结果
}
```

8.6.5.3 GetFirstLength

GetFirstLength 返回第一个 **Token** 的数据长度，当第一个 **Token** 不存在，返回 0

```
int CTonyXiaoMemoryQueue::GetFirstLength(void)
{
    int nRet=0;
    if(m_pHead)                             //如果 ICanWork 为否，m_pHead 必然为 null
    {
        nRet=m_pHead->m_nDataLength;        //取出第一个 Token 的长度
    }
    return nRet;
}
```

8.6.5.4 GetTokenCount

本函数是内联函数，在类声明中已经实现：

```
int GetTokenCount(void){return m_nTokenCount;}
```

8.6.5.5 PrintInside

与前述各类类似，MemQueue 提供队列内部各个 Token 的遍历打印方法，方便进行信令跟踪。本类使用了递归方式。

```
//私有函数，打印某一个指定的 Token，并且递归其后所有的 Token
void CTonyXiaoMemoryQueue::PrintAToken(STonyXiaoQueueTokenHead* pToken)
{
    if(!pToken) return;           //防御型设计，递归结束标志
    TONY_XIAO_PRINTF("Queue Token: pToken:%p, \
        Buffer=%p, Length=%d, m_pNext=%p\n", \
        pToken,                    //Token 指针
        pToken->m_pBuffer,          //数据缓冲区指针
        pToken->m_nDataLength,      //数据长度
        pToken->m_pNext);          //下一 Token 指针
    if(pToken->m_pBuffer)           //以二进制方式格式化输出业务数据
        dbg_bin(pToken->m_pBuffer,pToken->m_nDataLength);
    if(pToken->m_pNext)             //递归
        PrintAToken(pToken->m_pNext);
}
//打印所有的 Token 内容，公有函数入口
void CTonyXiaoMemoryQueue::PrintInside(void)
{
    if(!ICanWork()) return;       //防御性设计
    //输出队列关键信息
    TONY_XIAO_PRINTF("Queue: Token Count=%d, Head=0x%p, Last=0x%p\n",
        m_nTokenCount,            //Token 总数
        m_pHead,                  //队列头 Token 指针
        m_pLast);                 //队列尾 Token 指针
    if(m_pHead)                   //从队列头开始递归
        PrintAToken(m_pHead);
}
```

8.6.5.5 DeleteATokenAndHisNext

这是 CleanAll 函数需要调用的一个内部工具，删除一个 Token 以及其后所有的 Token。

```

//如果删除成功，返回真，否则返回假
bool CTonyXiaoMemoryQueue::DeleteATokenAndHisNext(
    STonyXiaoQueueTokenHead* pToken)          //纯 C 调用，需给定 Token 指针
{
    bool bRet=false;                          //准备返回值
    if(!ICanWork())                          //防御性设计
        goto CTonyXiaoMemoryQueue_DeleteATokenAndHisNext_End_Process;
    if(pToken->m_pNext)                        //注意，如果有 m_pNext，递归删除
    {
        DeleteATokenAndHisNext(pToken->m_pNext);
        pToken->m_pNext=null;
    }
    if(pToken->m_pBuffer)                      //开始删除本对象
    {
        m_pMemPool->Free(pToken->m_pBuffer); //向内存池释放数据缓冲区
        pToken->m_nDataLength=0;              //所有变量赋初值
        pToken->m_pBuffer=null;
    }
    m_pMemPool->Free((void*)pToken);           //向内存池释放 Token 头结构体缓冲区
    m_nTokenCount--;                           //Token 总计数器-1
    bRet=true;
CTonyXiaoMemoryQueue_DeleteATokenAndHisNext_End_Process:
    m_pLast=null;                             //删除动作后，由于链表尾部发生变化
                                              //加速因子失效，因此需要清空

    return bRet;
}

```

8.6.6 追加 AddLast

同 PopBuffer 类似，AddLast 是所有的队列模块最重要的方法，负责所有的数据压入队列操作。不过，请关注，MemQueue 的 AddLast，有维护加速因子 m_pLast 的责任。本类中，AddLast 不是一个独立函数，而是几个函数功能的集合。

8.6.6.1 GetAToken

在 AddLast 之前，我们需要先介绍一下 GetAToken 这个内部函数。我们知道，MemQueue 管理的链表节点，每个 Token 是采用纯 C 方式的结构体来管理，而没有采用 C++ 的类，这主要是方便使用内存池进行动态内存管理。

每个 Token，实际上分为两块内存区域，一块内存区域是 Token 的头结构，这也是从内存池上申请的动态内存，它管理着该 Token 相关的队列业务变量。其次是一块真实的数据内存块，里面存储业务数据。换言之，每次在 MemQueue 上增加一个 Token 节点，会申请两块动态内存区域。如图 8.6。

图 8.6: MemQueue 链表示意图

我们知道，这申请两块动态内存，分别初始化，这些工作也必须要有模块去完成，因此，笔者在 MemQueue 类中，设计了一个私有方法 GetAToken 来做这些初始化工作。请注意，这个函数创建并初始化 Token，返回指针，这是少有的不符合“谁申请，谁释放”原则的特例，主要是业务要求比较特殊。

```
//创建一个 Token 头内存区域，并初始化，返回 Token 指针
STonyXiaoQueueTokenHead* CTonyXiaoMemoryQueue::GetAToken(void)
{
    STonyXiaoQueueTokenHead* pToken=null; //准备返回值
    char* pTokenBuffer=null; //准备临时指针
    char szNameBuffer[256]; //准备说明文字缓冲区
    if(!ICanWork()) //防御性设计
        goto CTonyXiaoMemoryQueue_GetAToken_End_Process;
    //格式化说明文字，注意其中用到了 AppName
    SafePrintf(szNameBuffer,256,"%s::Token_Head",m_szAppName);
    //开始申请 Token 头内存块
    pTokenBuffer=(char*)m_pMemPool->Malloc( //请注意对说明文字的使用
        STonyXiaoQueueTokenHeadSize,szNameBuffer);
    if(!pTokenBuffer)
    { //申请失败，返回 null
        TONY_DEBUG("%s::GetAToken(): malloc new token fail!\n",m_szAppName);
        goto CTonyXiaoMemoryQueue_GetAToken_End_Process;
    }
    //强制指针类型转换，将申请到的二进制缓冲区指针转化成 Token 头结构体指针
    pToken=(STonyXiaoQueueTokenHead*)pTokenBuffer;
    pToken->m_nDataLength=0; //初始化过程，赋初值
    pToken->m_pNext=null;
    pToken->m_pBuffer=null;
    m_nTokenCount++; //Token 总数+1
CTonyXiaoMemoryQueue_GetAToken_End_Process:
    return pToken;
}
```

8.6.6.2 AddLast2ThisToken

由于 GetAToken 仅仅格式化了 Token 的头部，没有申请希望存储数据的内存块，因此，还需要 AddLast2ThisToken 这个函数做二次工作，继续进行业务数据的存储。本函数也是私有方法。从作用上看，本函数与 C++关系不大，更像一个 C 的链表操作函数。

```

//将数据保存到一个 Token，如果该 Token 已经保存数据，
//则按照链表顺序向下寻找，直到找到一个空的 Token，把数据放置其中。
//如果成功，返回数据长度，如果失败，返回 0
int CTonyXiaoMemoryQueue::AddLast2ThisToken(
    STonyXiaoQueueTokenHead* pToken,          //需要处理的 Token 头指针
    char* szData,                             //需要存储数据的指针
    int nDataLength)                          //需要存储的数据的长度
{
    int nRet=0;                               //准备返回值
    char szNameBuffer[256];                   //准备说明文字缓冲区
    if(!ICanWork())                          //防御性设计
        goto CTonyXiaoMemoryQueue_AddLast2ThisToken_End_Process;
    if(!pToken->m_pBuffer)
    {
        //如果本 Token 未包含有效数据，则保存到自己。
        SafePrintf(szNameBuffer, 256,         //格式化内存块说明文字
            "%s::pToken->m_pBuffer",
            m_szAppName);
        pToken->m_pBuffer=(char*)m_pMemPool->Malloc( //向内存池申请内存块
            nDataLength, szNameBuffer);
        if(!pToken->m_pBuffer)
        { //申请失败，报警
            TONY_DEBUG("%s::AddLast2ThisToken(): \
                ma lloc pToken->m_pBuffer fail!\n",
                m_szAppName);
            goto CTonyXiaoMemoryQueue_AddLast2ThisToken_End_Process;
        }
        memcpy(pToken->m_pBuffer, szData, nDataLength); //拷贝业务数据到内存块
        pToken->m_nDataLength=nDataLength; //填充 Token 头中的管理信息
        nRet=nDataLength; //给返回值赋值
        m_pLast=pToken; //保存加速因子(m_pLast 维护)
        goto CTonyXiaoMemoryQueue_AddLast2ThisToken_End_Process;
    }
    else
    {
        //保存在下家
        if(!pToken->m_pNext)
        {
            //如果指向下家的链指针为空，则利用 GetAToken 创建一个头挂接
            pToken->m_pNext=GetAToken();
            if(!pToken->m_pNext)
            { //创建失败报警
                TONY_DEBUG("%s::AddLast2ThisToken(): ma lloc \
                    pToken->m_pNext fail!\n",
                    m_szAppName);
                goto CTonyXiaoMemoryQueue_AddLast2ThisToken_End_Process;
            }
        }
        if(pToken->m_pNext) //递归调用
            nRet=AddLast2ThisToken(pToken->m_pNext, szData, nDataLength);
    }
}
CTonyXiaoMemoryQueue_AddLast2ThisToken_End_Process:
    return nRet;
}

```

8.6.6.3 AddLast

```
//返回添加的数据长度，失败返回0
int CTonyXiaoMemoryQueue::AddLast(
    char* szData,                //添加的数据指针
    int nDataLength,            //数据长度
    int nLimit)                  //最大单元限制，-1 表示不限制
{
    int nRet=0;                  //准备返回值，初值为0
    if(!ICanWork())              //防御性设计，条件不符合，直接返回0
        goto CTonyXiaoMemoryQueue_AddLast_End_Process;
    if(0>=nLimit)                //应用限制值
    { //这是无限制
        if(m_nMaxToken<=m_nTokenCount) //无限时，以m_nMaxToken 作为边界限制
            goto CTonyXiaoMemoryQueue_AddLast_End_Process;
    }
    else
    {
        //这是有限制
        if(nLimit<=m_nTokenCount) //如果有 nLimit，则使用这个参数限制
            goto CTonyXiaoMemoryQueue_AddLast_End_Process;
    }
    if(!m_pHead)
    {
        m_pHead=GetAToken();      //这是链头第一次初始化
        if(!m_pHead)
        {
            TONY_DEBUG("%s::AddLast(): ma lloc m_pHead fail!\n",
                m_szAppName);
            goto CTonyXiaoMemoryQueue_AddLast_End_Process;
        }
    }
    if(m_pLast)                  //加速因子开始起作用，如果有值，直接跳入
    {
        nRet=AddLast2ThisToken(m_pLast,szData,nDataLength);
    }
    else if(m_pHead)             //如果加速因子无值，按传统模式，遍历插入
    {
        nRet=AddLast2ThisToken(m_pHead,szData,nDataLength);
    }
    CTonyXiaoMemoryQueue_AddLast_End_Process:
    return nRet;
}
```

8.6.6.4 AddLast 小结

大家可能从上述代码中发现，笔者虽然以类方式实现了 MemQueue，但内部的链表管理并没有遵循 C++ 的模式，而是基本上以纯 C 方式实现了数据结构管理。这是因为我们明确说明，为了满足内存池管理需求，链表的基本节点，是纯 C 的结构体，而不是类。

其实很多时候，为了实现我们的业务，我们不一定要遵循一定使用 C++，或者一定使用 C 方式，**在商用工程实战中，对于技术，通常都是哪个合用用哪个，很少拘泥于形式。**笔者在此，希望各位读者了解，这种实用主义的态度，才是商用工程开发最核心的思想。

另外，关于加速因子 `m_pLast` 的管理，请各位读者仔细体会，代码很少，但效用巨大，很多时候，如果没有真实进行过底层数据结构开发的程序员，很少能理解到一些最基础，但最有效的优化手段。希望大家能深刻理解这个设计背后体现的深刻含义。

8.6.7 提取 GetAndDeleteFirst

同 `PopBuffer` 一样，`MemQueue` 也提供 `GetFirst`，`DeleteFirst`，`GetAndDeleteFirst` 等一系列接口，并且，出于对 api 的尊重，这些接口的所有函数构型完全一致，应用层可以很方便地选用。

8.6.7.1 GetFirst

`GetFirst` 提取第一个 `Token` 的数据，但不删除。和 `PopBuffer` 类似，本类的 `GetFirst` 也有两个重载。

```

//针对普通缓冲区进行提取，应用层保证缓冲区够大
//返回提取数据的真实长度（不一定是上层缓冲区的大小）
int CTonyXiaoMemoryQueue::GetFirst(char* szBuffer,int nBufferSize)
{
    int nRet=0;                //准备返回值变量
    //此处开始防御性设计
    if(!ICanWork())
        goto CTonyXiaoMemoryQueue_GetFirst_End_Process;
    if(!m_pHead)                //检查链表是否有数据
        goto CTonyXiaoMemoryQueue_GetFirst_End_Process;
    if(!m_pHead->m_pBuffer)      //检查第一个Token 是否有数据
    {
        TONY_DEBUG("%s::GetFirst(): m_pHead->m_pBuffer=null\n",
            m_szAppName);
        goto CTonyXiaoMemoryQueue_GetFirst_End_Process;
    }
    if(m_pHead->m_nDataLength>nBufferSize) //检查给定的缓冲区是否足够
    {
        TONY_DEBUG("%s::GetFirst(): m_pHead->m_nDataLength > nBufferSize\n",
            m_szAppName);
        goto CTonyXiaoMemoryQueue_GetFirst_End_Process;
    }
    memcpy(szBuffer,m_pHead->m_pBuffer,m_pHead->m_nDataLength);    //拷贝数据
    nRet=m_pHead->m_nDataLength;    //返回值赋值
CTonyXiaoMemoryQueue_GetFirst_End_Process:
    return nRet;
}
//这是针对 Buffer 类对象的提取函数
int CTonyXiaoMemoryQueue::GetFirst(CTonyBuffer* pBuffer)
{
    if(!ICanWork()) return 0;                //防御性设计
    if(!pBuffer) return 0;
    //调用上一函数完成功能
    return pBuffer->BinCopyFrom(m_pHead->m_pBuffer,m_pHead->m_nDataLength);
}

```

8.6.7.2 DeleteFirst

DeleteFirst，无条件删除第一个 Token，请注意，这和 DeleteATokenAndHisNext 不一样，不删除后续 Token，反而要将第二 Token 的指针赋给 m_pHead。

```

//删除第一个Token，有可能失败，因为队列可能为空，失败返回假，成功返回真
bool CTonyXiaoMemoryQueue::DeleteFirst(void)
{
    bool bRet=false; //准备返回值
    STonyXiaoQueueTokenHead* pSecond=null; //准备指向第二Token的临时指针
    if(!ICanWork()) //防御性设计
        goto CTonyXiaoMemoryQueue_DeleteFirst_End_Process;
    if(!m_pHead)
        goto CTonyXiaoMemoryQueue_DeleteFirst_End_Process;
    //注意，先提取First中保留的Socond指针，做中间保护
    pSecond=m_pHead->m_pNext;
    //m_pHead的m_pNext赋为空值，这是割裂与队列其他元素的连接关系
    m_pHead->m_pNext=null;
    //然后调用DeleteATokenAndHisNext完成删除，由于上面的割裂，因此不会影响其他Token
    bRet=DeleteATokenAndHisNext(m_pHead);
    if(bRet)
    {
        m_pHead=pSecond; //重新给m_pHead挂接第二Token
        //完成对原有First的挤出工作
        //此时，原有Second变为First
        if(!m_pHead) //这里有一个特例，
            m_pLast=null; //如果本次删除把该队列删空，
        //需要清除加速因子，避免下次操作失败
    }
    else
    {
        //如果删除失败（这种情况一般不太可能，属于保护动作，不会真实执行）
        //将队列恢复原状，起码不至于在本函数内部造成崩溃，或内存泄漏，但打印报警
        TONY_DEBUG("%s::DeleteFirst(): delete m_pHead fail!\n",
            m_szAppName);
        m_pHead->m_pNext=pSecond; //删除失败，还得恢复回去
    }
}
CTonyXiaoMemoryQueue_DeleteFirst_End_Process:
    return bRet;
}

```

8.6.7.3 GetAndDeleteFirst

由于已经存在上述两个逻辑，因此，GetAndDeleteFirst变得很简单。只需要简单执行先取后删逻辑即可。

```

int CTonyXiaoMemoryQueue::GetAndDeleteFirst(char* szBuffer,int nBufferSize)
{
    int nRet=GetFirst(szBuffer,nBufferSize);
    if(nRet) DeleteFirst();
    return nRet;
}
int CTonyXiaoMemoryQueue::GetAndDeleteFirst(CTonyBuffer* pBuffer)
{
    int nRet=GetFirst(pBuffer);
    if(nRet) DeleteFirst();
    return nRet;
}

```

8.6.8 PopBuffer 相关操作

这部分功能函数，主要是针对商用数据传输工程的特例，实现信令打包的弹出和推入。

8.6.8.1 信令编组弹出

信令编组弹出，主要针对一个上层程序准备的缓冲区，不是进行单条信令的弹出，而是利用 PopBuffer 类的粘合功能，格式化一个 PopBuffer 缓冲区，实现批量信令的格式化弹出。

弹出的原则有如下几条：

- 1、如果队列被弹空了，则返回。
- 2、如果队列没有被弹空，但 PopBuffer 满了，无法插入，也弹出，注意，此时不能使用 GetAndDeletFirst，否则弹出的数据很不容易插回队列头。

```

//返回弹出的数据总长度，但请注意，不是 PopBuffer 的 AllBytes，仅仅是业务数据长度和
int CTonyXiaoMemoryQueue::PopFromFirst4TonyPopBuffer(
    CTonyPopBuffer* pPopBuffer)          //需传入 PopBuffer 指针
{
    int nRet=0;                          //准备返回值变量
    if(!ICanWork())                      //防御性设计
        goto CTonyXiaoMemoryQueue_PopFromFirst4TonyPopBuffer_End_Process;
    if(!m_pHead)
        goto CTonyXiaoMemoryQueue_PopFromFirst4TonyPopBuffer_End_Process;
    if(!m_pHead->m_pBuffer)              //这也是防御性设计，检索 First 是否有数据
    {
        TONY_DEBUG("%s::PopFromFirst4TonyPopBuffer(): \
            m_pHead->m_pBuffer=null\n",
            m_szAppName);
        goto CTonyXiaoMemoryQueue_PopFromFirst4TonyPopBuffer_End_Process;
    }
    if(!m_pHead->m_nDataLength)          //防御性设计，检索 First 数据尺寸是否合法
    {
        TONY_DEBUG("%s::PopFromFirst4TonyPopBuffer(): \
            m_pHead->m_nDataLength=0\n",
            m_szAppName);
        goto CTonyXiaoMemoryQueue_PopFromFirst4TonyPopBuffer_End_Process;
    }
    if(!pPopBuffer)                     //检查给定的 PopBuffer 是否合法
    {
        TONY_DEBUG("%s::PopFromFirst4TonyPopBuffer(): pPopBuffer=null\n",
            m_szAppName);
        goto CTonyXiaoMemoryQueue_PopFromFirst4TonyPopBuffer_End_Process;
    }
    //没有使用 GetAndDeleteFirst 之类的函数，而是直接访问结构体内部变量
    //就是预防如果 PopBuffer 满了，弹出来的数据不好处理，这样也减少无谓的动态内存分配
    nRet=pPopBuffer->AddLast(m_pHead->m_pBuffer,m_pHead->m_nDataLength);
    if(m_pHead->m_nDataLength!=nRet)
    {
        //这是 buffer 装满了
        goto CTonyXiaoMemoryQueue_PopFromFirst4TonyPopBuffer_End_Process;
    }
    if(!DeleteFirst())                  //删除 First
    {
        TONY_DEBUG("%s::PopFromFirst4TonyPopBuffer(): DeleteFirst fail\n",
            m_szAppName);
        goto CTonyXiaoMemoryQueue_PopFromFirst4TonyPopBuffer_End_Process;
    }
    if(m_pHead)                        //注意此处的递归逻辑，继续向下弹
        nRet+=PopFromFirst4TonyPopBuffer(pPopBuffer);
CTonyXiaoMemoryQueue_PopFromFirst4TonyPopBuffer_End_Process:
    return nRet;
}

```

很多时候，业务层并不关心给出的信令包是否采用 PopBuffer 打包，而是简单给出一段报文缓冲区，又本类进行填充，因此，这个公有接口函数，就是为了适应普通缓冲区情况。


```

//向普通缓冲区，以 PopBuffer 方式打包信令
int CTonyXiaoMemoryQueue::PopFromFirst(char* szBuffer,int nBufferSize)
{
    int nCopyBytes=0; //准备返回值
    //准备一个 PopBuffer 对象，注意，粘合作用。
    CTonyPopBuffer PopBuffer(szBuffer,nBufferSize);
    if(!ICanWork()) //防御性设计
        goto CTonyXiaoMemoryQueue_PopFromFirst_End_Process;
    if(m_pHead)
    { //调用上一函数实现打包
        nCopyBytes=PopFromFirst4TonyPopBuffer(&PopBuffer);
        if(nCopyBytes) //如果弹出了数据（数据长度!=0）
            //取 PopBuffer 的 AllBytes 作为返回值
            nCopyBytes=PopBuffer.GetAllBytes(); //这就是实际需要发送的字节数
    }
CTonyXiaoMemoryQueue_PopFromFirst_End_Process:
    return nCopyBytes;
}

```

8.6.8.2 信令编组推入

信令编组推入队列，是弹出的反操作。一般说来，网络传输的发送方，会从自己的队列中弹出一个信令编组，然后发送到远端接收方。接收方收到以后，一般有两种可能：

- 1、如果是同步动作逻辑，则可能直接使用 PopBuffer 的 GetAndDeleteFirst 函数，逐条弹出，并解析执行。
- 2、如果接收方本身是中继服务器，属于传输层业务，并不理解业务逻辑，则很可能使用本小节的功能，将信令包整体打入另外一条异步传输缓冲队列 MemQueue，然后等待下一次发送，此时，就需要用到信令编组推入。

之所以这么设计，是因为笔者在实做时，遇到一个很费解的设计，必须如此优化。我们来看一下例子：

原则上，中继服务器收到报文后，往往会将其直接转发，这是常规逻辑，即透明传输。但是，在实际的网络数据传输工程中，情况往往并不这么理想。我们来看这个模型：

图 8.6: 中继转发示例

我们知道，当 A、B、C 三台客户端都在不同内网中时，它们之间的通信往往需要一台公网中继转发服务器来协助。在某种业务逻辑下，我们假定 A 和 B 需要传输信息给 C，因此，需要做如下工作：

1、A 和 B 分别主动向中继服务器发起了一次 TCP 连接，并在连接成功后，将其设置为上行链路，即数据传输方向是 A、B--->中继服务器。

2、C 主动向中继服务器发起了一次 TCP 连接，并在连接成功后，将其设置为下行链路，即数据传输方向是中继服务器--->C。

我们前文说过，如果信令的传输质量需求是确定传输，则必须考虑在传输中设置 Ack 信号，接收方需要给发送方一个显式的发送成功标志。但此时，传输的包交换率很低，一般说来，每秒钟，也就 40~50 个报文左右，如果要保证传输速率，必须考虑信令打包，这也就是 PopBuffer 设计的由来。

但是，此时的中继服务器上，我们会发现一个有趣的现象。比如我们假定，A 每秒传输 30 个报文到中继服务器，每个报文中平均 20 条信令，这样没有问题，中继到 C 这个链路能够满足传输需求，但假如 B 也按照这个速率发送数据，则每秒钟，在 C 的下行链路上，报文个数的传输需求就变成了 $30 \times 2 = 60$ 个报文，此时已经超过了带 Ack 的传输链路的传输速率。如果此时还有其他网络角色希望传输信令给 C，则这个问题会更加严重。

这意味着，每秒钟需要传输给 C 的报文，远远超过了 C 下行链路的承载能力，则必然在中继服务器上发生淤塞。如果这类业务属于连续业务，则必然会导致中继服务器缓冲队列溢出，此时，要么丢失信令，要么中继服务器崩溃。

因此，这里有个很重要的结论：“**中继服务器必须将收到的信令做二次打包，接收的报文个数与发送的报文个数无任何必然关联性，收发逻辑完全异步进行。**”

这说明，即使不理解业务的中继服务器，收到信令编组报文后，也必须拆解入队列，下次发送时，按照队列中信令情况，重新编组，打包发送。

这里给大家提示一点报文和信令的差别：在笔者看来，信令，是业务层，为了完成一个业务逻辑，发送的业务命令。而报文是传输层，每次收发，所发送或接收的所有二进制数据的集合，一个报文，既可以只包含一个信令，也可以包含多个。

正是因为上述原因，笔者才设计出信令编组推入队列的功能，帮助中继服务器完成上述动作。

前文我们知道，PopBuffer 有一个 MoveAllData 的功能，能构建一个循环，将 PopBuffer 内部所有数据，逐一弹出，用回调函数的形式，通知上层业务层。这里我们利用这个功能完成。我们首先看看接口函数：

```
//将受到的一段 PopBuffer 格式化的信令编组，推入 MemQueue 的队列末尾
//返回推入的数据字节常数，失败返回 0
int CTonyXiaoMemoryQueue::Push2Last(char* szData,int nDataLength)
{
    int nRet=0;
    CTonyPopBuffer PopBuffer(szData,nDataLength,false);    //此处粘合
    if(!ICanWork())    //防御性设计
        goto CTonyXiaoMemoryQueue_Push2Last_End_Process;
    //开始调用 PopBuffer 的功能，实现遍历循环
    //请注意回调函数和 this 指针参数
    PopBuffer.MoveAllData(PushDataCallback,this);
CTonyXiaoMemoryQueue_Push2Last_End_Process:
    return nRet;
}
```

其中的 PushDataCallback 是本类准备的回调函数，这里，我们复习一下回调函数的声明，在 C++ 的类声明中，回调函数必须是静态函数。

```
static bool PushDataCallback(char* szData,int nDataLength,void* pCallParam);
```

现在，我们来看看实现：

```
//回调函数，返回 bool 值，如果返回 false，遍历循环将退出，信令可能会丢失。
bool CTonyXiaoMemoryQueue::PushDataCallback(
    char* szData,    //拆解出来的信令地址
    int nDataLength,    //信令数据长度
    void* pCallParam    //代传的参数指针，就是前面的 this
)
{
    //笔者的习惯，一般类内回调函数，均以 this 指针作为透传参数
    //在回调函数内部，一般称之为 pThis，以区别于普通成员函数的 this
    //回调函数一进入，第一件事就是强制指针类型转换，创建 pThis，方便调用。
    CTonyXiaoMemoryQueue* pThis=
        (CTonyXiaoMemoryQueue*)pCallParam;
    //此处为调用实例，pThis 指针就是 this，因此，调用的是本对象的 AddLast
    int nRet=pThis->AddLast(szData,nDataLength);
    //成功，返回 thre，遍历继续，直到数据遍历结束
    if(nDataLength==nRet) return true;
    else
    {
        //失败，可能是队列满了，返回 false，终止遍历，
        pThis->TONY_DEBUG("%s::PushDataCallback(): I am full!\n",
            pThis->m_szAppName);
        return false;
    }
}
```

这里有一个很重要的细节，请大家关注：大家可以看到，由于 MemQueue 有 Token 数量上限，因此可能会满，AddLast 之后会失败，此时，本函数会直接返回 false，抛弃信令。

这个设计是基于一个设计理念，传输层不负责维护数据传输的确定性，或者说，部分维护，每次传输，传输层有 Ack 校验的义务，但在自己资源不够，无法满足需求时，允许抛弃信令。避免传输层崩溃。

由此得知，确定传输的实现，不仅仅是传输层的义务，业务层一旦设计一笔确定传输业务，必须也设计相应的校验机制，层层把关，最终达到目标。

8.6.9 文件保存相关操作

文件保存相关操作，主要应对的是 7*24 小时的服务程序，在偶尔下线检修，服务器重启等维护状况时，必须将队列中的信令保存至磁盘，在下次启动时，再逐一读回，避免丢失队列中的待发数据。

这和下文的文件型队列不一样，MemQueue，主要还是工作于内存中，设计中强调高性能，高吞吐量，此处仅仅是提供一种队列内数据的永固保存手段。而文件型队列主要以磁盘文件作为缓冲，强调大队列容量，以及大数据存储量需求，主要应对长时间断网造成的信令大量淤塞问题。

请注意，由于队列数据到磁盘保护，已经是数据的最终极保护形式，如果读写文件再出错，程序本身已经没有任何办法，只能抛弃信令和数据。因此，此处不再做更高的安全校验策略，磁盘出错，即报警，然后抛弃信令返回即可。

8.6.9.1 Write

Write 主要分为两个函数，大家可能注意到，MemQueue 有一点不如 PopBuffer，PopBuffer 的数据是连续的内存区域，因此，很容易整体读写。但 MemQueue 由于使用链表管理，数据是串在一起的零散数据块，必须用遍历循环来逐块读写数据。

大家可能注意到，本类中凡是涉及到针对每个 Token 的操作，都是基于纯 C 实现链表的方式，即参数传入 Token 的指针。

```

//将指定 Token 的数据写入文件
void CTonyXiaoMemoryQueue::WriteAToken2File(
    STonyXiaoQueueTokenHead* pToken,          //指定的 Token 指针
    FILE* fp)                                  //指定的文件指针
{
    if(!ICanWork()) return;                    //防御性设计
    if(!fp) return;
    if(!pToken) return;
    if(!pToken->m_pBuffer) return;
    if(!pToken->m_nDataLength) return;
    //写入磁盘，由于磁盘写入，通常发生在服务器即将退出期间，这已经是数据保护最大的努力
    //此时磁盘写入再出现 bug，没有任何办法再保护数据，只能丢失信令
    //因此，此处不再做校验，如果磁盘写入失败，即丢失信令
    //先写入每个 Token 的数据长度
    fwrite((void*)&(pToken->m_nDataLength), sizeof(int), 1, fp);
    //再写入每个 Token 的实际数据
    fwrite((void*)pToken->m_pBuffer, sizeof(char), pToken->m_nDataLength, fp);
    if(pToken->m_pNext)                          //递归到下一 Token
        WriteAToken2File(pToken->m_pNext, fp);
}
//这是公有方法入口函数
void CTonyXiaoMemoryQueue::Write2File(char* szFileName) //需给出文件名
{
    FILE* fp=null;
    if(!ICanWork()) return;                    //防御性设计
    if(!m_pHead) return;
    if(!GetTokenCount()) return;              //如果队列为空，直接返回
    fp=fopen(szFileName, "wb");                //打开文件
    if(fp)
    {
        //首先，将队列控制信息写入磁盘，以便读回时直接使用
        fwrite((void*)&m_nTokenCount, sizeof(int), 1, fp);
        //开始调用上述递归函数，开始逐 Token 写入
        WriteAToken2File(m_pHead, fp);
        fclose(fp);                          //关闭文件
    }
}

```

8.6.9.2 Read

读和写很类似，仅仅是动作的反向。在本函数实例中，大家可以看到关于内存池的 ReMalloc 的用法。请注意，这个读入工作，并不破坏队列中原有数据，仅仅是追加到末尾。

```

//从一个磁盘 Dump 文件中读入数据，返回读入的 Token 总数
int CTonyXiaoMemoryQueue::ReadFromFile(char* szFileName)
{
    FILE* fp=null;
    int n=0;
    int i=0;
    int nReadTokenCount=0;
    int nDataLength=0;
    char* pTempBuffer=null;
    char szNameBuffer[256];
    if(!ICanWork()) //防御性设计
        goto CTonyXiaoMemoryQueue_ReadFromFile_End_Process;
    SafePrintf(szNameBuffer,256, //构建内存申请说明文字
        "%s::ReadFromFile::pTempBuffer",m_szAppName);
    //申请临时缓冲区，由于临时缓冲区需要在循环中多次 ReMalloc，因此，开始只申请 1Byte 即可
    pTempBuffer=(char*)m_pMemPool->Malloc(1,szNameBuffer);
    fp=fopen(szFileName,"rb"); //打开文件
    if(!fp) //失败则跳转返回
        goto CTonyXiaoMemoryQueue_ReadFromFile_End_Process;
    //读入队列控制头，即 TokenCount 信息
    n=fread((void*)&nReadTokenCount,sizeof(int),1,fp);
    if(1>n)
    {
        TONY_DEBUG("%s::ReadFromFile(): read token count fail!\n",m_szAppName);
        goto CTonyXiaoMemoryQueue_ReadFromFile_End_Process;
    }
}

```

```

for(i=0;i<nReadTokenCount;i++)                //开始逐一读出各个 Token
{
    //首先读出当前 Token 的数据长度
    n=fread((void*)&(nDataLength),sizeof(int),1,fp);
    if(1>n)
    {
        TONY_DEBUG("%s::ReadFromFile(): %d / %d, read data length fail!\n",
            m_szAppName,i,nReadTokenCount);
        goto CTonyXiaoMemoryQueue_ReadFromFile_End_Process;
    }
    if(0>nDataLength)
    {
        TONY_DEBUG("%s::ReadFromFile(): %d / %d, nDataLength=%d < 0!\n",
            m_szAppName,i,nReadTokenCount,nDataLength);
        goto CTonyXiaoMemoryQueue_ReadFromFile_End_Process;
    }
    //调用 ReMalloc, 根据读入的 nDataLength 重新分配缓冲区, 以便读入数据
    pTempBuffer=(char*)m_pMemPool->ReMalloc(
        pTempBuffer,                //原有的临时缓冲区地址
        nDataLength,                //新的数据长度
        false);                     //由于这个缓冲区马上被覆盖
                                    //因此无需拷贝旧数据

    if(!pTempBuffer)
    {
        TONY_DEBUG("%s::ReadFromFile(): rema lloc pTempBuffer fail!\n",
            m_szAppName);
        goto CTonyXiaoMemoryQueue_ReadFromFile_End_Process;
    }
    //准备缓冲区成功, 开始读入该 Token 数据
    n=fread((void*)pTempBuffer,sizeof(char),nDataLength,fp);
    if(nDataLength>n)
    {
        TONY_DEBUG("%s::ReadFromFile(): read data fail!\n",m_szAppName);
        goto CTonyXiaoMemoryQueue_ReadFromFile_End_Process;
    }
    //读入成功, AddLast 添加到最后
    if(!AddLast(pTempBuffer,nDataLength)) break;
}
CTonyXiaoMemoryQueue_ReadFromFile_End_Process:    //出错跳转标签
if(pTempBuffer)                                  //清除临时缓冲区
{
    m_pMemPool->Free(pTempBuffer);
    pTempBuffer=null;
}
if(fp)                                            //关闭文件
{
    fclose(fp);
    fp=null;
}
return nReadTokenCount;                          //返回读入的 Token 总数
}

```

8.6.10 线程安全封装类

当我们完成上述逻辑，MemQueue 的基本功能已经齐备，下面我们按照资源锁的理念，开始为其构造一个线程安全锁的封装类 CTonyMemoryQueueWithLock，这里也可以看出笔者的命名习惯，一个类的线程安全封装类，就是其类名后面添加 WithLock，一目了然。

由于 MemQueue 的使用特殊性，通常用于服务器的高频任务机，或者高速转发缓冲队列，因此，笔者为其设计使用单写多读锁，以提升并发读效率。

8.6.10.1 线程安全封装类声明

线程安全封装类，原则上都是利用聚合实现，而不会采取继承，因为聚合我们可以很方便地控制暴露在外的公有方法，并且开发实现很容易。

请注意封装类的特性：

- 1、仅仅实现所有公有方法，原类的私有方法无需管理。
- 2、所有实现的公有方法函数，构型与原类对应函数应该一模一样，方便应用层做无缝切换。
- 3、封装类内部尽量不要实现任何功能，仅仅做调用关系的透传。

```
//MemQueue 的线程安全封装类
class CTonyMemoryQueueWithLock
{
public:
    CTonyMemoryQueueWithLock(CTonyLowDebug* pDebug,
        CTonyMemoryPoolWithLock* pMemPool,
        char* szAppName,
        int nMaxToken=TONY_CHAIN_TOKEN_MAX);
    ~CTonyMemoryQueueWithLock();
public:
    bool ICanWork(void); //对应各种公有方法名
    int AddLast(char* szData,int nDataLength);
    int GetFirst(CTonyBuffer* pBuffer);
    int GetFirst(char* szBuffer,int nBufferSize);
    int GetFirstLength(void);
    int GetTokenCount(void);
    int GetAndDeleteFirst(char* szBuffer,int nBufferSize);
    int PopFromFirst(char* szBuffer,int nBufferSize);
    int Push2Last(char* szData,int nDataLength);
    void CleanAll(void);
    bool DeleteFirst(void);
    void Write2File(char* szFileName);
    int ReadFromFile(char* szFileName);
    void PrintInside(void);
private:
    CTonyMemoryPoolWithLock* m_pMemPool;
    CTonyXiaoMemoryQueue* m_pQueue; //MemQueue 的聚合对象
    CMultiReadSingleWriteLock m_Lock; //线程安全锁，为了保证效率，
    //此处使用了单写多读锁
    char m_szAppName[TONY_APPLICATION_NAME_SIZE]; //保存的 AppName
};
```

8.6.10.2 构造函数和析构函数

线程安全封装类的构造函数和析构函数，原则上仅负责实例化封装对象以及摧毁它，尽量不要涉及任何新的业务功能。

//构造函数，请注意，参数和MemQueue 完全一样

```
CTonyMemoryQueueWithLock::CTonyMemoryQueueWithLock(
    CTonyLowDebug* pDebug,
    CTonyMemoryPoolWithLock* pMemPool,
    char* szAppName,
    int nMaxToken)
{
    //保存 AppName
    SafeStrcpy(m_szAppName,szAppName,TONY_APPLICATION_NAME_SIZE);
    m_pMemPool=pMemPool; //保存内存池指针，析构函数要用
    m_pQueue=new CTonyXiaoMemoryQueue(pDebug, //实例化封装对象
        pMemPool,m_szAppName,nMaxToken);
    if(m_pQueue)
    {
        char szNameBuffer[256];
        //如果实例化成功，则试图在内存池注册它，以实现指针管理
        //注册前，先利用 AppName 构造说明文字
        SafePrintf(szNameBuffer,256,"%s:m_pQueue",m_szAppName);
        m_pMemPool->Register(m_pQueue,szNameBuffer);
    }
}
//析构函数，摧毁封装对象爱那个
CTonyMemoryQueueWithLock::~CTonyMemoryQueueWithLock()
{
    m_Lock.EnableWrite(); //此处是带写函数，因此使用写锁
    {
        if(m_pQueue)
        {
            m_pMemPool->UnRegister(m_pQueue); //这部反注册很重要，否则内存池报警
            delete m_pQueue; //摧毁对象
            m_pQueue=null; //习惯，变量摧毁后立即赋初值
        }
    }
    m_Lock.DisableWrite();
}
```

8.6.10.3 纯读函数族

通常，队列面临大量的查询操作，这类操作，都具有纯读特性，因此，可以使用单写多读锁的“读锁”来操作，以提升并发特性。

ICanWork 的功能前面已经介绍了，主要用于判断各种运行条件是否齐备，封装对象是否已经实例化成功等，同时调用封装对象内部的 ICanWork 做进一步精确校验。

```

bool CTonyMemoryQueueWithLock::ICanWork(void)
{
    if(!m_pMemPool) return false;
    if(!m_pQueue) return false;
    bool bRet=false;
    m_Lock.AddRead();
    {
        bRet=m_pQueue->ICanWork();
    }
    m_Lock.DecRead();
    return bRet;
}

```

GetFirst 是典型的读操作

```

int CTonyMemoryQueueWithLock::GetFirst(CTonyBuffer* pBuffer)
{
    int nRet=0;
    m_Lock.AddRead();
    {
        nRet=m_pQueue->GetFirst(pBuffer);
    }
    m_Lock.DecRead();
    return nRet;
}

int CTonyMemoryQueueWithLock::GetFirst(char* szBuffer,int nBufferSize)
{
    int nRet=0;
    m_Lock.AddRead();
    {
        nRet=m_pQueue->GetFirst(szBuffer,nBufferSize);
    }
    m_Lock.DecRead();
    return nRet;
}

```

GetFirstLength 也是典型的读操作

```

int CTonyMemoryQueueWithLock::GetFirstLength(void)
{
    int nRet=0;
    m_Lock.AddRead();
    {
        nRet=m_pQueue->GetFirstLength();
    }
    m_Lock.DecRead();
    return nRet;
}

```

GetTokenCount 是典型的读操作

```

int CTonyMemoryQueueWithLock::GetTokenCount(void)
{
    int nRet=0;
    m_Lock.AddRead();
    {
        nRet=m_pQueue->GetTokenCount();
    }
    m_Lock.DecRead();
    return nRet;
}

```

Write2File 虽然定名为写，但就 **MemQueue** 而言，还是纯读操作，读出内存中数据写入磁盘。

```

void CTonyMemoryQueueWithLock::Write2File(char* szFileName)
{
    m_Lock.AddRead();
    {
        m_pQueue->Write2File(szFileName);
    }
    m_Lock.DecRead();
}

```

PrintInside 是纯粹的读操作

```

void CTonyMemoryQueueWithLock::PrintInside(void)
{
    m_Lock.AddRead();
    {
        m_pQueue->PrintInside();
    }
    m_Lock.DecRead();
}

```

8.5.10.4 带写函数族

带写函数只在运行过程中，对队列数据有写动作，可能会破坏原有数据，因此，需要使用写锁做强制串行保护。

AddLast 必然是写操作

```

int CTonyMemoryQueueWithLock::AddLast(char* szData,int nDataLength)
{
    int nRet=0;
    m_Lock.EnableWrite();
    {
        nRet=m_pQueue->AddLast(szData,nDataLength);
    }
    m_Lock.DisableWrite();
    return nRet;
}

```

DeleteFirst 和 **GetAndDeleteFirst** 也是写操作

```

bool CTonyMemoryQueueWithLock::DeleteFirst(void)
{
    bool bRet=0;
    m_Lock.EnableWrite();
    {
        bRet=m_pQueue->DeleteFirst();
    }
    m_Lock.DisableWrite();
    return bRet;
}

int CTonyMemoryQueueWithLock::GetAndDeleteFirst(
    char* szBuffer,
    int nBufferSize)
{
    int nRet=0;
    m_Lock.EnableWrite();
    {
        nRet=m_pQueue->GetAndDeleteFirst(szBuffer,nBufferSize);
    }
    m_Lock.DisableWrite();
    return nRet;
}

```

Pop 和 Push 都涉及队列内部数据的该表，是写操作

```

int CTonyMemoryQueueWithLock::PopFromFirst(char* szBuffer,int nBufferSize)
{
    int nRet=0;
    m_Lock.EnableWrite();
    {
        nRet=m_pQueue->PopFromFirst(szBuffer,nBufferSize);
    }
    m_Lock.DisableWrite();
    return nRet;
}

int CTonyMemoryQueueWithLock::Push2Last(char* szData,int nDataLength)
{
    int nRet=0;
    m_Lock.EnableWrite();
    {
        nRet=m_pQueue->Push2Last(szData,nDataLength);
    }
    m_Lock.DisableWrite();
    return nRet;
}

```

CleanAll 是写操作

```

void CTonyMemoryQueueWithLock::CleanAll(void)
{
    m_Lock.EnableWrite();
    {
        m_pQueue->CleanAll();
    }
    m_Lock.DisableWrite();
}

```

ReadFromFile，是从磁盘中读入数据，写入内存队列，因此，实际上是写操作。

```

int CTonyMemoryQueueWithLock::ReadFromFile(char* szFileName)
{
    int nRet=0;
    m_Lock.EnableWrite();
    {
        nRet=m_pQueue->ReadFromFile(szFileName);
    }
    m_Lock.DisableWrite();
    return nRet;
}

```

8.7 关于队列的小结

队列可以说是商用并行系统至关重要的组成部分，在并行计算体系中，在数据传输系统中，在很多的场合，队列都起着重要的作用。

本章我们简要介绍了几种常用的内存队列，使读者对程序员自行实现数据传输相关的业务队列有了一个基础的了解。但我们应该清楚地知道，队列的使用情形远远不止这么多。

比如，在很多高可靠性的系统中，我们可能更加需要一种存储于文件中的队列，一些数据分发型的队列等。这些，在以后的章节中，我们会补充介绍。

但是，不管属于哪种队列，请各位读者相信，队列的共性是想通的，笔者在此做出一个简单的总结，请大家参考：

1、队列基于其“先入先出”的特性，总是提供 AddLast 和 GetAndDeleteFirst 这两个基本方法，供所有业务层使用。

2、并行计算中，异步转同步，一般使用锁或者队列解决，对于实时业务场合，如高传输速率，以速度优先进行优化的流媒体，实时语音或视频场合，使用锁可能更方便一些，而在很多高可靠性保证场合，如信令传输，分布式数据库操作，队列几乎是唯一的选择。

3、网络中的角色，由于其硬件设备特性，担负的业务流程特性，往往吞吐量并不一致，这就出现了很多网络角色之间的“速度差”，这很类似于一台 PC 计算机中，各个部件，如 CPU、内存和硬盘这类设备之间的“速度差”，调整这类速度差的唯一方法，就是使用队列做缓冲。

4、通常情况下，一个逻辑的网络链路，为了稳定、安全实现其功能，在物理实现中，除了一个基本的 socket 之外，都还有一个发送队列与其绑定，以适应各种异步的数据交换情况。

5、很多时候，我们需要利用队列，把网络角色之间的速度差距导致的不匹配，转化为一段内存空间的增大和减小，这种时间差到空间差的转化，通常是网络程序能和谐运行的基本手段。如上文中，多个角色向一个角色发送数据，中继服务器需要在其对应的逻辑单元使用队列，以协调这类收发速率差。

6、在很多网络服务器设计中，我们把收发行为区别为两个逻辑，这是解决并行干涉，提升服务器吞吐量的基本手段，而这种调整，通常需要使用队列作为中间缓冲。

7、队列天生的数据存储能力，是解决网络临时中断时，信令在某个网络角色的临时暂存手段，很多自适应的网络系统，主要依靠队列来无缝适应网络通畅和网络中断的各种情况，以对外提供稳定的服务品质。

综上所述，笔者提醒各位读者，如果希望开发商用并行系统，对队列的构建和使用应该非常熟悉，需要仔细揣摩队列的各种特性，各种优化方式，以便在以后的工作中，减少风险和 bug。

第9章 时间片管理

- 9.1 多线程与单线程开发的差异
 - 9.1.1 单任务操作系统运行程序的特点
 - 9.1.2 任天堂游戏机中断机制
 - 9.1.3 利用中断实现多任务
 - 9.1.4 多任务操作系统运行程序的特点
 - 9.1.5 多任务操作系统运行程序的机制
 - 9.1.6 多任务运行环境的世界观
- 9.2 多任务操作系统常见线程操作
 - 9.2.1 线程相关变量
 - 9.2.2 线程函数声明
 - 9.2.3 线程函数启动
 - 9.2.4 MIN_SLEEP
 - 9.2.5 线程操作总结
- 9.2 线程池
 - 9.2.1 线程池的来源和需求分析
 - 9.2.2 线程池的设计原理
 - 9.2.3 线程池的基本数据结构
 - 9.2.4 线程池的类设计说明
 - 9.2.5 构造函数和析构函数
 - 9.2.4 管理者线程
 - 9.2.5 服务者线程
 - 9.2.6 注册函数
 - 9.2.7 线程池小结
- 9.4 任务池
 - 9.4.1 任务池的原理分析
 - 9.4.2 任务池的需求和设计
 - 9.4.3 任务池的基本定义说明
 - 9.4.4 任务池的类声明
 - 9.4.5 构造函数和析构函数
 - 9.4.6 管理者线程
 - 9.4.7 服务者线程
 - 9.4.4 任务注册接口
 - 9.4.4 任务池的小结及实现示例
- 9.5 任务池的运行体
 - 9.5.1 简化运行态
 - 9.5.2 任务描述工具类
 - 9.5.3 任务池运行体的设计原理
 - 9.5.4 任务池运行体的类声明
 - 9.5.5 StartTask
 - 9.5.6 StopAll 和 PrintInfo
 - 9.5.7 任务执行线程回调
 - 9.5.8 任务池运行体小结及调用示例
- 9.6 时间片小结

第 9 章 时间片管理

本章中，笔者将给大家展示，如何实现一个线程池，以及从线程池引申出任务池管理，进一步地，我们将了解多任务系统以时间片为核心的任务机的运行原理，以及讨论如何利用这些知识，使应用程序员掌握并行处理中，时间片的基本管理方法，实现高效的系统设计。

9.1 多线程与单线程开发的差异

笔者可能是较早接触多任务开发的程序员之一。

在 1995 年，笔者加入一家游戏公司，开始为任天堂 8 位游戏机（红白机）写插卡游戏程序。这是一款很古老的游戏机，使用 6502 8 位 CPU，只有 2k RAM，内存编址为 0~0xFFFF(65535)。根据需要，最高位的 16k 内存，可以映射到游戏卡中的某块内存，并运行其中的程序，游戏卡可以做得很大，比如，512k，然后一某种块切换方式工作，每次调用其中 16k 的程序。

在这个系统上的开发工作，使笔者第一次对多任务环境有了清醒的认识。

9.1.1 单任务操作系统运行程序的特点

我们知道，常规的程序，比如一段 C 语言程序，总是以 main 开始，然后顺序向下执行，一直执行到 main 最后的 return，然后程序结束。我们称之为一个执行生命周期。不过，在多任务操作系统和单任务操作系统看来，这个过程有点不同。

在单任务操作系统（比如 DOS）看来，用户在键盘上敲入一个文件名并回车，操作系统就开始执行一段流程，从磁盘上读入一段代码，放置到内存中合适的地方，然后，操作系统跳到该段代码处开始运行，直到应用程序执行完，操作系统重新收回执行权，重新等待用户输入，以便开始下一个任务。

单任务操作系统最大的特点，就是在用户程序运行时，会把整个计算机的所有权，交给用户程序，操作系统本身丧失了对系统的控制权，用户应用程序，掌控所有的资源：内存、外设、所有的 CPU 时间片，甚至，所有的中断操作。

这样设计的坏处显而易见，万一用户的应用程序有 bug，出现死循环，或者出现了内存非法访问，整个计算机系统随即崩溃，必须重新启动计算机。同时，操作系统没有任何优先级保护，连操作系统自身的代码，都是裸露在内存中，应用程序可以任意读写访问，因此，很可能一个应用程序的 bug，导致操作系统自身被破坏，也会导致系统崩溃，被迫重启。大家现在知道 PC 机的 Reset 键是怎么来的了吧？最古老的 DOS 无法满足自身安全运行的需求，被迫硬件上设置一个中断，重启计算机。

这些缺陷，甚至被有心人利用，刻意编写恶意代码，修改操作系统的部分功能为己所用，并传播破坏，这就是计算机病毒的前身。

同时，用户的应用程序，也不是每时每刻都在工作，比如一个字处理程序，大多数时间，在等待用户的输入，由其控制的 CPU 在空转等待，计算机的内存也由于这个原因，每次只能提供给一个应用程序使用，哪怕这个应用程序只使用 5k 内存，运行时也独占全部的计算机内存空间，这些，造成了巨大的资源浪费。

因此，单任务操作系统，其提供的应用程序运行环境，是不安全的，也是极其浪费的，所以随着计算机的发展，单任务操作系统逐渐被淘汰。

在笔者进行任天堂游戏机的开发时，情况比 DOS 下运行程序还坏，因为这台计算机（这也算计算机）上，没有任何操作系统，每次加电启动，硬件直接通过某个内存单元的值，直接跳转到我们的机器码程序中，开始运行。

以后的内存块切换，时间片管理这些操作系统基本功能，全部由程序员自己完成。因此，这台游戏机提供的是一个比 DOS 还不如的纯单任务操作环境。

这就带来一个很大的问题，比如我们需要做一个射击游戏，我们的飞机在空中飞行，不断有多个敌机来袭，子弹满屏幕飞舞，玩家操作自己的飞机，灵活地躲避，设计，最终完成任务通关，这是大多数射击类游戏经常出现的场景。

这里我们来看，同一时间段，玩家的飞机，多个敌机，还有几十发，上百发子弹同时满屏飞舞，每个角色有其各自的飞行轨迹，但又能彼此碰撞互动，这种效果是怎么做出来的呢？

显然，这里面需要用到多任务开发的技术，即，每个角色（我方飞机，敌机，子弹）计为一个任务，然后利用多任务机制，实现并发运行，才能造就上述效果。

但是，在这么简陋的计算机系统上，显然，我们并没有多任务操作系统的支持，因此，我们首先要实现一套多任务运行机制，才能开始上述设计，这也是笔者在此写出这个例子的原因，希望能够帮助各位读者从底层理解多任务执行机制的工作原理。

9.1.2 任天堂游戏机中断机制

我们知道，在单任务操作系统下，我们同一时间段，只有一个执行生命体，这就是说，原则上，同一个时间段，只有一个计算在进行。这就给笔者带来一个很大的问题：游戏通常是多个动画角色在同时运动，以达到丰富多彩的游戏效果，如何通过一个计算，使多个数据块表示的动画角色，都同时移动起来？

其实这已经涉及到多任务业务需求了，即业务需求上，同一个时间段，有多个计算需要并发进行。在单 CPU 计算机系统上，这实际上是不可能的。

经过学习，笔者发现任天堂这台计算机，虽然非常简陋，但其有个非常优秀的特点，它根据屏幕扫描线的回扫动作，提供出 IRQ 和 NMI 两路硬件中断信号。如图 9.1。



图 9.1: 任天堂游戏机的中断示意图

如图所示, 我们知道, CRT 电视机, 是使用 X 射线轰击荧光粉发光, 由于 X 射线很细, 一个屏幕又很宽, 通常扫描线是一行一行地扫描轰击, 描述出一个个的亮点或暗点, 最终扫完一个屏幕, 形成一幅图像。

另外, 当扫描速度足够快, 每秒钟我们能扫描超过 25 帧画面时, 由于电影的视觉残留影像原理, 我们就能看到动画。学过计算机图形学的读者, 应该都知道这个道理。

任天堂游戏机, 巧妙地利用了这个原理, 它的硬件在控制内存数据到屏幕的扫描过程中, 当扫描完一行, 扫描焦点回到屏幕左边, 准备开始第二行扫描时, 硬件会发出一个 IRQ 中断。当扫描完一整屏画面, 扫描焦点从屏幕右下角回到左上角, 准备开始新的一屏扫描时, 硬件会发出一个 NMI 中断。就这么简简单单的两个中断, 开启了任天堂游戏机丰富多彩的图形动画游戏世界。

我们知道, 中国的交流电频率是 50Hz, 美国和日本交流电略有不同, 是 60Hz。一般说来, 电视机等动画系统, 其扫描频率是交流电频率是差不多的, 也是 50Hz 左右。(电视机上是隔行扫描, 因此, 一屏仅仅扫描一半的点数, 两次扫描一整屏, 因此, 电视机实际上是 25Hz 左右的扫描频率)

任天堂游戏机的图形信号发生器, 基本上也是基于上述原理工作, 每秒扫描 50 次屏幕, 这就很好计算了, 我们由上面的原理可以得知, 每秒钟, 至少会发生 50 次 NMI 中断, 平均 20ms 一次。

另外, 由于行扫描是匀速进行, 任天堂游戏机的屏幕分辨率是 256*240, 即横向 X 方向上有 256 个点, 纵向 Y 方向上, 有 240 条扫描线, 这样经过简单计算得知, 每 20ms, 完成 240 次扫描线, 也就是发生 240 次 IRQ 中断。这样计算一下: 每次 IRQ 中断的时间, 差不多 0.083ms 左右。

这说明, 如果我们在任天堂游戏机程序中, 拦截了 IRQ 和 NMI 中断, 使其跳入到我们的程序中执行, 那么, 每隔 0.083ms 左右, 我们的 IRQ 程序会被执行一次, 每隔 20ms 左右, 我们的 NMI 程序会被自动执行一次。

当然，这里面有个很大的限制，就是每次我们执行的代码，不能太长，比如 IRQ 中断拦截程序，如果太长，执行时间超过了 0.083ms，下一次 IRQ 又发生了，此时，就会造成中断程序重入，前一段 IRQ 程序没有执行完，就被第二次 IRQ 程序打断，重新开始执行，此时，程序运行混乱，系统就会崩溃。

这需要一点计算。从资料上得知，任天堂游戏机的 6502 CPU 时钟频率是 1MHz，也就是说，每秒会发生 100 万个时钟周期，因此，每次 IRQ 中断，差不多 83000 个时钟周期左右。每次 NMI 中断，差不多 20 万时钟周期左右。

由于 6502CPU 的指令，平均时钟周期在 1~3 周期/指令之间，因此，一般说来，每次 IRQ 中断，可以最多执行 2 万条指令左右。换言之，只要我们设计的 IRQ 中断程序，每次只执行 20000 条左右的指令即返回，程序运行就是安全的，不会出现上述的崩溃现象。（NMI 由于时间较长，一般不会出现重入，此处不再计算。）

值得关注的是，这个 20000 条指令的数量并不算很多，稍微一个大点的循环，就可能超过这个限制，因此，笔者在开发中，曾多次被迫将一个大循环，切割成多个小循环来完成，以规避可能的中断重入。

最典型的例子是大场景刷屏，即游戏进入一个新的关卡，需要切换整个屏幕的画面时，这个切换动作一般要分为三个周期完成，每次刷三分之一屏幕，否则就会因为中断重入，屏幕花屏，程序崩溃。

9.1.3 利用中断实现多任务

可能部分读者会觉得上面的论述非常难以理解，这和我们写程序有什么关系？那我们换一种说法，可能帮助大家理解：

1、任天堂游戏机，如果按照传统的程序开发方式，同一时间段，只有一个计算在进行，因此，很难实现多任务开发，也就无法完成游戏中，多个角色同时动作的需求。

2、但是，任天堂游戏机，利用硬件给了我们两路中断，一路 IRQ，每 0.083ms 一次，一路 NMI，每 20ms 一次，其中，IRQ 每次可以执行大约 20000 条代码左右。通常，为了保证高速性，我们使用 IRQ 中断。

3、一个 IRQ 中断，会导致我们的一段代码执行，当我们完成一笔业务之后，可以返回，等待下一次中断的产生，这个过程，我们可以视为一个“时间片”。

4、如果我们仔细规划我们的中断体系，预设多个任务，每次 IRQ 中断，即每个“时间片”，我们依次轮询，处理其中一个业务，则我们可以模拟出一个多任务并行系统。经过计算，在 1 秒钟内，我们有 $240 \times 50 = 12000$ 个时间片可以使用，那么理论上，我们可以同时处理 12000 个任务，这些任务每秒钟都会有一次运行机会。看起来，就好像有 12000 个程序在同时运行。

5、事实上，笔者开发的游戏程序，通常一启动，就进入一段死循环，常规的执行生命体，除了一些必要的初始化动作，几乎不做任何事，所有的游戏业务逻辑，全部在中断中完成。

这就是利用中断实现多任务仿真运行环境的基本原理，任天堂游戏机是这样设计的，在 PC 机上，多任务操作系统的设计者，差不多也是按照这个原理在设计。即利用硬件中断，巧妙地打断常规执行流程的进行，实现任务时间片划分和切换，进而构造多任务并行执行效果。

虽然笔者使用汇编开发的任天堂游戏机程序，但是，这里我们还是可以很轻松地用 C 语言模拟一下一个游戏核心中断处理逻辑。

这里我们假设一个游戏，最多全屏并发 8 个动画角色（精灵），大家可以看到，下面这段逻辑实现了依次轮询方式。

总的来说，下面的实例，最多划分 9 个时间片，前 8 个时间片，每片处理一个精灵，最后一个时间片，处理系统事务，如果系统事务过于繁杂，可以内部再使用这段代码的类似结构，进行进一步的时间片划分。

事实上，一个任天堂的游戏程序，核心逻辑全部是这类时间片划分函数的嵌套关系，构成了中心循环体系。

```
void BreakFunc(void)
{
    static int nSpriteIndex=0; //精灵索引
    switch(nSpriteIndex)
    {
        case 0:
            Sprite_Func_0();          //精灵处理程序
            nSpriteIndex++;            //下一时间片处理下一个精灵
            break;
        case 1:
            Sprite_Func_1();          //精灵处理程序
            nSpriteIndex++;            //下一时间片处理下一个精灵
            break;
        case 2:
            Sprite_Func_2();          //精灵处理程序
            nSpriteIndex++;            //下一时间片处理下一个精灵
            break;
        case 3:
            Sprite_Func_3();          //精灵处理程序
            nSpriteIndex++;            //下一时间片处理下一个精灵
            break;
        case 4:
            Sprite_Func_4();          //精灵处理程序
            nSpriteIndex++;            //下一时间片处理下一个精灵
            break;
        case 5:
            Sprite_Func_5();          //精灵处理程序
            nSpriteIndex++;            //下一时间片处理下一个精灵
            break;
        case 6:
            Sprite_Func_6();          //精灵处理程序
            nSpriteIndex++;            //下一时间片处理下一个精灵
            break;
        case 7:
            Sprite_Func_7();          //精灵处理程序
            nSpriteIndex++;            //下一时间片处理下一个精灵
            break;
        default:
            Some_Idle_Func();          //滚屏，碰撞检测，其他服务等
            nSpriteIndex=0;
            break;
    }
}
```

笔者也曾经做过 PC 游戏的开发试验，情况和上述设计差不多，PC 机上，一般不使用视频中断，这是因为 PC 机的显示卡有多种型号，CGA、EGA、VGA、SVGA 等等，这些显示子系统的点阵数，扫描频率各不相同，因此，如果仿造任天堂游戏机，使用视频中断作为任务切换中断，则很难计算出通用的时间片长度，最大指令执行数等关键参数。

不过，PC 机有另外的工具，IBM 在设计 PC 机时，为该计算机系统设计了很多种中断，其中，最有用的就是时钟中断，通常情况下，程序员在 DOS 下开发游戏程序，如 DOOM 等，都是拦截时钟中断来实现时间片切换，进而实现多角色，多任务的游戏效果。

我们再深入一点，目前所有的 32 位多任务操作系统，其最核心的时间片切换，其实利用的也是这个原理，即以某个定时的硬件时钟中断来切换任务，达到多任务并行工作的目的。

总结：通常在单 CPU 下的多任务运行环境，其核心就是利用一定的中断机制，来定时打断某些正在执行的任务，去执行另外的任务，如此周而复始，模拟同一时间段，多个程序并行运行的结果。当然，多 CPU 也可以实现这个机制。

中断体系，更多的类似一个循环体，这个循环是无限死循环，且循环动作的控制，不是由我们的 while、for、jmp 等具体的软件指令来完成的，而是利用系统周而复始的定时中断，使中断处理子程序不断被调用而构成的。巧妙地利用这一点，可以构建出几乎所有的游戏循环，当然，应用程序循环也可以利用这个方式构建。

9.1.4 多任务操作系统运行程序的特点

当然，前面我们讲述游戏的例子，其中实现的多任务运行环境还很原始，它只能调用我们预先开发好的程序片段，实现特定的功能，并且，由于中断时间的限制，每个程序片段，只能执行经过精确计算的有限代码，以防止中断重入导致的崩溃。

而 Windows、Linux 等 32 位多任务操作系统，明显比上述模型复杂得多，它们设计的目的，是为了适应任何情况，其运行的任务，是动态的，根据用户启动的进程数随时在变化，也是随着运行程序的不同而不同的，同时，由于用户进程可能开辟大量的线程，实际执行情况更加复杂，不过，最核心的执行原理还是同前面讲述的一致。

不过，要实现一个完整的多任务操作系统，我们还是有很多细节需要关注的。

9.1.4.1 时间片切换问题

在设计这类多任务操作系统时，我们可以看到，其实每个操作系统，一定要仿照上述机制，内部实现一个任务机，以便以一定策略分配时间片，即一个中断到来时，可以根据一定的数据表，确认跳到哪段代码中执行。

但这就带来一个问题，我们知道，在任天堂游戏机中，代码片段是我们自己设计的，因此，我们会在大约 20000 条指令之前，结束一小段工作，返回，将代码执行权放弃，以便下次中断的到来。

但多任务操作系统显然需要适应用户用各种语言编写的各种程序，一般的解释系统还好办，但诸如 C 和 C++ 这类典型的编译程序，一般程序的开发是顺序执行的流程，从头到尾，用户程序员不可能替操作系统考虑何时该切换时间片，将执行权放回给操作系统的问题。

此时，操作系统如果要构建一个多任务运行环境，势必需要利用另外一些中断，来打断当前时间片的执行，避免一个时间片任务太长，别的任务无法执行，最后又变回单任务操作系统。通常，操作系统会利用 CPU 的某些条件中断，在一定时间里，打断正在执行的时间片，以进行任务切换。

9.1.4.2 最大时间片长度问题

这里面还有另外一个问题，就是最大时间片长度，前文我们知道，在中断中运行的程序，每次运行周期执行的代码数不能太长，避免出现中断重入。这个我们已经得知，一般操作系统会利用 CPU 上的一些中断，来实现定时强行打断时间片的问题。

不过，很多时候，操作系统并不建议这么做，因为如果每次都等到超时来打断时间片，操作系统自身就没有太多的时间来做一些必要的管理工作，因此，一般建议用户的应用程序，在完成一小段任务后，礼貌地将运行权返回给操作系统，不要每次将时间片占满。

通常这个工作，由编译器代为完成。无论是 VC 还是 gcc，都带有多线程库，（如 VC 下的/MT 编译参数），这是以适应多线程方式开发的 C 和 C++ 基本库，其接口 api 与单线程完全一样，以达到兼容编译的目的，但内部针对多线程做了很多优化。

我们前文说明的，在线程中执行循环，需要在每个循环周期内嵌入一个 Sleep 调用，休眠一下，这在编译器看来，就知道目前的程序逻辑，已经基本告一段落，如果要切换时间片，这个时间最合适。因此，就可能在此嵌入一些代码，将操作权及时放给操作系统，帮助操作系统在此时收回执行权。

因此，我们这里可以看出单任务和多任务运行环境下，Sleep 休眠函数的不同之处。

1、在单任务环境下，Sleep 就是 CPU 空转，等待一段时间，返回继续执行程序。

2、在多任务环境下，Sleep 变成一种应用程序向操作系统的承诺，就是承诺下面一段时间内，本程序不需要自己时间片。操作系统完全可以把这些分配给本程序的时间片，转给其他程序使用。

这是前文介绍的，循环内使用 Sleep 的真实根源。多任务程序员应该养成这个习惯，在自己的任务完成一个小片段后，主动调用 Sleep，帮助操作系统回收时间片，以便降低系统 loading，使整个操作系统运行环境更加合理地分配时间片资源。

9.1.4.2 粗暴地打断执行带来的问题

但是，即使我们程序员养成了调用 Sleep 的好习惯，很多时候，我们的任务可能还是会超过一个时间片的限制，比如说一次复杂的公式计算工作，一次超大规模的数据库检索遍历等，在这些业务逻辑导致的大型任务中，需要执行很多高密度的计算，因此不适宜添加 Sleep。

这就说明，操作系统也仅仅能建议应用程序员，尽量养成使用 Sleep 的习惯，但是无法绝对杜绝，总有一些任务很大，超过操作系统预设的时间片大小，因此，操作系统将被迫在一定时间后，中断这段代码的执行，硬性把时间片分配给其他任务使用。

而此时，大家可以看到，操作系统的中断行为，对应用程序而言，是一种“粗暴的打断”，很可能在其某个指令执行到一半的时候，被打断，等过一段时间再来执行。

而如果此时，计算操作的数据区，是其他线程也要访问的数据区时，由于一个动作被打断后，其他针对该数据区的操作可能会在中断期间发生，因此，争用发生了，这就是多任务程序开发中，变量争用问题，这就导致了锁的应用，程序员必须使用线程安全锁，来临时保护某一块数据区，以免在被强行打断期间，数据被其他执行生命体修改，导致 bug。

我们前文花了大量的篇幅来介绍线程安全锁的使用，同时也介绍了多个锁的应用实例，其核心根源，就是为了避免这种粗暴打断带来的恶果。

9.1.5 多任务操作系统运行程序的机制

由于上面我们讲了这么多时间片相关的知识，这里我们可以总结一下：

1、多任务操作系统是通过时间片的切分，通过把不同的时间片，分配给不同的任务，来模拟多任务并行的。

2、虽然有多核 CPU，以及多 CPU 体系的计算机，但是，多任务操作系统本身，和 CPU 数目并无直接关系。

由此我们得知，多任务操作系统，在执行程序时，有一个执行生命体的概念。

从操作系统角度看，所谓执行生命体，就是指一个任务，被注册到多任务运行环境中，环境承认它的合法存在，为其分配逻辑内存空间，并能在自己的任务调度表中，分配一定的时间片给它，使它的计算能获得 CPU、内存等资源，得到真实地执行。操作系统保证它能被执行，但它执行的效率，取决与操作系统根据优先级算法分配给它的时间片的多少。操作系统既不保证它的执行速度，也不保证它在什么时间点一定会得到执行。

从任务本身看，执行生命体，就是指其被从磁盘上调出，获得了逻辑虚拟的进程内存空间，并获得了一定的 CPU 使用权。一个任务，只有以执行生命体形式存在，才算真正获得了运行权，它就可以预期，在将来某个时刻，自己的代码，一定能获得 CPU 执行。但这个时刻是什么时候，以及自己能获得时间片的多少，并不是它能控制的。

通常情况下，我们把多任务操作系统的执行生命体，称之为一个任务。

这样，我们可以总结出单任务运行环境 and 多任务运行环境下最大的差异性：

单任务环境下，操作系统一旦开始执行某个应用程序，即将 CPU 和内存的控制权，交给这个任务代码，此时，操作系统退居后台，除了必要的中断响应服务，基本上操作系统已经丧失了对计算机系统的控制权，直到应用程序执行完毕返回。从时间轴上看，任务执行情况为下图所示：

图 9.2：单任务操作系统运行应用程序

多任务操作系统下，任何时候，操作系统都具有整个计算机系统的控制权，所有的任务在其控制下运行，每个任务的执行，都是不连续的，片段的。并且，时间片与执行生命体之间的对应关系是乱序的，完全根据操作系统的优先级调度策略来计算确定，任务本身，

图 9.3：多任务操作系统以时间片方式运行多个应用程序

不可预期自己在将来某个时间片，一定可以获得执行。如下图：

因此，这里我们可以得出多任务操作系统和单任务操作系统最大的核心差异。

单任务操作系统，运行应用程序，是执行权的转移，操作系统把系统几乎所有的计算资源，全部转交给应用程序代码，开始执行。从某种意义上讲，单任务操作系统，其代码地位并没有多大优势，在执行的地位上说，它和应用程序是平级的，因此，应用程序的崩溃，可以导致操作系统的崩溃。

多任务操作系统，运行应用程序，则是一次注册行为，操作系统内部有一个任务链表，所有位于该链表上的任务，都有可能获得运行权，进程、线程的启动，仅仅是在该链表上，注册一个新的任务，使之获得 CPU 执行权，能在将来的某个时刻获得执行。但归根到底，

多任务操作系统的进程或线程，这类执行生命体，是操作系统通过调度算法，动态把时间片分配给他，获得执行的。操作系统永远掌握最高运行优先权。

这里给大家可以分享一点多任务操作系统的知识：最开始的时候，多任务操作系统，比如 Windows3.1、Windows95 等多任务操作系统，由于机制所限，并没有完全地控制时间片，一旦把 CPU 交给某个执行生命体，如果该应用程序的逻辑是执行高密度计算，不做返回，则仍然很可能把操作系统挂死。这也是 Windows95 等系统一直不够稳定的根源。

但幸运的是，从 WindowsNT, Windows2000 等纯 32 位操作系统开始，实现了完全的“抢先式多任务”，即通过调用 CPU 内的一些中断机制，操作系统在某些编写得不太好的应用程序，不肯释放 CPU 使用权时，仍然有能力在某个时间点，强行打断应用程序的执行，实现时间片的切换。这直接改善了操作系统的稳定性，各位读者可能能感觉到，随着 WindowsNT 架构的成熟，目前使用的 WindowsXP 等操作系统，已经很少出现蓝屏和死机现象，本质上就是这个原因。

至于 Linux，由于其全仿 Unix 架构体系特征，因此，从一开始就是一个完整意义上的多用户、多任务操作系统，对应用程序执行“抢先式多任务”模式，因此，Linux 相对于 Windows 来说，一般要稳定一些。

当然，Windows 的 RING0 级内核代码，仍然不允许任意打断执行，而在 2.4 版的 Linux 之前，内核也是不可打断的，即使是 2.6 版之后的 Linux，内核允许打断，但由于大量自旋锁等机制的存在，其实打断的效果仍然不好。因此，Linux 下的“抢先式多任务”，基本上可以认为，也仅仅对应用程序有效。

由于仍然存在很多不可打断的代码，Windows 和 Linux 操作系统，都不是严格意义上的实时操作系统。

9.1.6 多任务运行环境的世界观

从前文可知，由于多任务操作系统和单任务操作系统，有这么大的差异性，笔者认为，我们有必要从世界观的角度，彻底重新认识一下多任务运行环境。

9.1.6.1 这个世界是大家的

一个 CPU，同一时间点，只能执行一个计算，这是我们都知道的，因此，一旦一个程序开始以独占方式启动，它就拥有这台计算机系统所有的资源：外设、内存、CPU、等等。这也是传统单任务操作系统的开发思维，**只要我的程序开始运行，这个世界就是我的。**

但是，多任务操作系统，巧妙地在“时间点”这个基础上，创造了“时间段”这个概念，每个点只能执行一个计算，这个我们同意，但是，我可以在不同的时间点，执行不同的计算，那么，在宏观角度看来，在一个时间段内，比如说 1 秒钟内，有多个任务被同时进行计算。

这说明，多任务操作系统下，**即使我的程序开始运行，这个世界也不完全是我的，我们必须考虑合作者的需求，要有礼貌，要懂得规避。**

当然，现代意义上的 32 位多任务操作系统，已经为应用程序做了很多事情，最关键的，就是创造了进程私有内存空间这个概念，起码从计算最重要的资源内存这个角度来说，一个应用程序基本上看到的是自己的私有数据，只要计算不涉及外设等公共设备，一般说来，应用程序员还是可以以单任务环境方式开发自己的程序，无需过多考虑资源争用带来的冲突。

但是，随着多线程程序开发的兴起，这个原则再次被打破，对于同一个进程而言，所有的线程共享同一个内存空间，我们知道，线程和进程一样，都是操作系统承认的合法执行生命体，都有 CPU 时间片的使用权，这必然带来了资源争用，也带来了锁的使用需求。

因此，多任务程序员在进行程序开发时，必须时刻牢记“礼貌”和“规避”。随时记得这个世界不仅仅是自己一个人的，还有多个兄弟姐妹在共享，对任何资源的独占方式运行，都是不合理的，也是非法的。

笔者在 C 和 C++ 无错化程序设计方法中，特别强调变量的私有化，强调锁的使用，并建议大家利用好 C++ 类的私有化封装，其核心思想正是来自于此，既然多线程的程序开发成为必然，而各个线程共享进程内存空间，如果程序员不做好变量的私有化以及安全访问问题，那么，多任务程序的开发，将会变成一种灾难。

9.1.6.2 注册的真实含义

在前文中，我们提到了大量“注册”这个概念，那我们来讨论一下，什么叫“注册”。

我们首先来复习一下，在内存池中我们实现的注册手段。在 `CMemoryRegister` 这个类中，我们对外提供方法，应用程序可以把一个内存指针，连同其说明文字，临时存放在其中，并可以做增改删操作，并能查询某个指针是否存在。

如果把这些行为抽象出来，其实我们可以发现，提供“注册”方法的对象，更像一个数据库，对外提供增改删和查询的方法，目的是将一些关键信息，临时存放在某个数据中心，当需要时，应用程序可以查询使用其中的数据。

操作系统在管理当前活动任务，就是所有在运行的执行生命体时，其实也采用了类似的机制。一般说来，不管是 Windows 还是 Linux 系统，内部都有一个到几个活动任务链表，这个链表可以对外提供增改删操作，也可以实现遍历和查询。

我们启动一个进程，或者程序中启动一个线程，其实就是在这些链表上，增加一个元素，当操作系统下次遍历链表，以便实现时间片分配时，就可以看到新增加的这个元素，



图 9.4：多任务操作系统任务链表

并为其分配时间片和执行权，于是，新的程序获得了真实的执行。如图所示：

当然，当程序逻辑执行完毕，如一个线程函数完成任务，退出时，会有代码自动从链表上将自己的注册信息删除，操作系统下次再也看不到这个任务，也就不会再有新的时间片分配的。

这就是注册的真实含义，也是多任务操作系统，启动一个进程或线程任务，真实所做的事情。

但是，这里有一个重要细节请注意，这个链表是整个操作系统执行任务机的核心，因此，必然受到最高级别的锁保护，在操作系统级，有一些 CPU 级的自旋锁，是绝对不能打断的，因此，注册动作可能会失败，即我们调用了启动线程的函数，但线程可能没有真实地启动起来。

不管是 Windows 系统还是 Linux 系统，都有一个禁忌，一般说来，如果一个应用程序连续不断地高速启动线程，则很可能线程启动会失败。即程序看起来启动动作执行正确了，但是，线程函数并没有得到真实地执行。主要的原因，就是高频注册新的任务，会导致任务机过于繁忙，很多时候，注册动作可能会因为锁的问题，没有得到真实的执行，就超时返回了。

前文说过，根据笔者的经验，一般说来，线程的启动，间隔 250ms 会比较安全，低于这个数值，则很可能导致“死线程”的出现，即一个线程，看起来注册启动成功，但最后没有真实的执行生命体存在，程序也没有真正获得执行。

9.1.6.3 C 和 C++ 角度理解多任务

既然我们讨论了多任务环境下，进程和线程是以一个“注册”动作启动的。那么，我们是不是可以思考一下，这个“注册”动作，究竟填充了哪些数据到系统的链单元中呢？

我们以 Windows 为例，笔者设想，至少进程句柄（Handle）应该有，操作系统会以此查询该进程和线程，使用的是哪个虚拟内存空间。如果是线程，还应该查询到该函数的入口地址，以及这个函数当前执行点等信息。其他不论，但这些信息必须有，否则，操作系统无法实现时间片调度和执行的功能。

在 C 和 C++ 中，都有回调函数的概念，我们知道，C 里面，函数的本质还是指针，指向一段代码，我们所谓的调用函数行为，其实就是跳到该指针指向的代码去执行。这就说明，如果我们设计一段程序，完全可以以一个函数指针作为参数，必要时，跳到这个指针指向的代码去执行，至于这个指针，可以在编译器完全不确定，等运行期动态确定。

这其实就是 C 和 C++ 语言，回调函数的本质。

那么，我们反过来看，多任务操作系统的进程和线程本质，是不是就是回调函数？

当我们启动一个进程，操作系统会寻找这个进程相关的程序代码中，main 这个函数指针，并跳转到该处开始执行，程序于是启动，而线程就更简单了，由于 C 和 C++ 的线程，本身就是函数，只需要将这个函数的指针，注册到操作系统的任务执行机中，下次会自动会跳到该处执行。

这个概念非常重要，请各位读者一定要记住：“**多任务操作系统，任务就是回调函数**”，笔者随后向大家展示的线程池、任务池，都是基于这一理念创建的。

9.1.6.4 多任务并行计算的本质

经过了这么多的讲解，相信各位读者，对多任务并行计算的本质，有了一个初步的了解。

我们知道，一台计算机，只要一启动，随着时间的流逝，时间片总是无穷无尽地产生，不管有没有任务，这个时间片总是不断产生，又不断过去。一个多任务操作系统，掌握的最大资源，其实就是无穷无尽的时间片。

所谓任务的执行，其实就是在系统的某个任务链表上，注册一些信息，标明了一个任务代码的起始地点，那么，操作系统就会为其分配时间片，使其获得执行。

在任天堂游戏机中，我们实际上遇到的是一个完全硬件抢先式多任务运行环境，但是，由于没有操作系统的概念，它只能执行有限功能，并且，功能的安全执行，依赖程序员的精心设计和计算。

而现代意义上的多任务操作系统，理论上可以适应任何应用程序，其根源就是它采用了注册任务机的形式，以回调函数形式将 CPU 执行资源，分配给不同的任务，因此，才造就了目前几乎所有的程序，都可以在 Windows 和 Linux 操作系统上实现的现状。

这类开发大量接口，开放时间片资源，以回调形式适应任何任务的机制，就是多任务并行计算的本质。

9.2 多任务操作系统常见线程操作

前面我们一起理解了多任务操作系统下，如何看待线程的问题，这是多任务并行开发的基础知识。当然，我们的目的，是利用这些知识，开发出商用数据传输工程合用的线程池，以至于后面的任务池，方便以后的工程开发。

但是，在进行所有这些线程相关开发前，我们有必要先学习一下基本的线程概念，开发方法，以及跨平台进行多线程开发的一些基本技术。

9.2.1 线程相关变量

无论是 Windows 系统还是 Linux 系统，对于一个线程，都需要一些特殊的变量类型以及定义，以此界定这个线程的地址，代号，错误码等。这些变量非常重要，但在笔者看来，又很不重要。这个我们需要详细说明。先请看看声明：

```
#ifndef WIN32
#define THREAD_HANDLE          //线程句柄变量
#define THREADID unsigned      //线程 ID
#define THREADCREATE_ERROR 0    //线程启动错误码
#else // not WIN32 (Linux)
#define THREAD pthread_t        //线程句柄变量
#define THREADID unsigned       // Posix Threads 不使用
#define THREADCREATE_ERROR -1   //线程启动错误码
#endif
```

一个线程，操作系统要识别并管理它，一般说来，都必须有一个线程的句柄，这个概念在 Windows 下很常见，简单说就是一个类似指针的数据，在内存中标明了一个对象的数据位置，方便操作系统予以管理。

对于线程而言，Windows 和 Linux 都有一个句柄，由于我们的程序需要跨平台兼容，因此，这里我们统一使用宏 THREAD 来做定义，方便以后开发跨平台程序。

在 Windows 下，这个句柄就是 HANDLE，这个类型做过 Win32API 的读者朋友一定很熟悉，就是 Windows 下基本的句柄类型，可以表示任何 Windows 环境下的对象（注意，这不是 C++ 的对象，是 Windows 系统的操作数据单元）。

Linux 下是 pthread_t 这个变量类型，最终是一个整数，Linux 和 Unix 系统，很多情况下，都是以整数作为一个操作单元的索引。

值得一提的是，Windows 下比 Linux 多一个线程相关变量类型，就是线程 ID，我们使用 THREADID 来表示。

但上述信息并不重要，我们只需要知道，当我们以 begin_thread 开启一条线程，可以获得两个变量，THREAD 和 THREADID 即可，任何时候，我们可以通过这两条变量来找到这个线程。

笔者要说，这个变量很重要，又很不重要，为什么这么说呢？

我们知道，当一个线程在正常运行时，它是独立的执行生命体，自行执行其代码，与外界交互很少。如果需要数据交互，一般也是在线程安全锁的保护下，通过共享变量，共

享数据区，队列等手段和其他线程（如主线程）进行数据交互，由于这些数据一般是应用程序自行定义好的，不需要操作系统提供手段实现交互。因此，数据交互，一般用不到上述两个变量。

那还有什么地方用呢？一般说来，当应用程序开启线程，是有责任关闭线程才退出的，不管 Windows 还是 Linux 操作系统，都提供了类似 `kill_thread` 之类的方法函数，就是从外部杀掉一个线程，当调用此方法的时候，需要使用上述两个变量。

但是，前面我们说过，线程操作有个很严厉的原则：“**永远不要从外部杀死一个线程，否则容易造成资源泄漏**”，这在 C 和 C++ 无错化程序设计方法中，已经成为一个铁的原则。换言之，`kill_thread` 这个方法，在我们的商用并行工程中，一般是永不使用的。

这样看来，即使我们启动一个线程，获得了上述两个变量，但其实没有任何用途，操作系统提供了这个机制，来帮助我们在将来某个时刻杀死线程，但我们不会使用。

因此，出于节约内存原则，以及不保存无意义数据，避免 bug 的原则，笔者的线程管理模块，原则上不保存上述两个变量，线程启动即抛弃，以后利用其他方法，使线程温和退出。

至于 `THREADCREATE_ERROR`，这是不同操作系统在执行相关线程操作时返回的错误码，Windows 下为 0，Linux 为 -1。

9.2.2 线程函数声明

我们再看看线程函数声明，首先，我们需要确定一个细节，“**线程就是操作系统级的回调函数，操作系统为每个注册的线程函数提供唯一一次运行机会**”。

这个概念很重要，虽然我们用 C 和 C++ 开发软件，里面用到了很多类，对象等面向对象概念，但在操作系统看来，线程，就是函数，是一个过程，用户一旦注册启动一根线程，首先需要提供一个回调函数，这个回调函数是按照操作系统规定的构型声明的，这点绝对不能变的。

既然是回调函数，就必须按照我们前文说明的方法定义，即要么定义成全局函数，要么定义成 C++ 类的静态成员函数，不能是普通函数，因为操作系统调用该函数时，没有 `this` 指针提供。

线程实际上有两种退出方式，一种是前文说明的外部线程以 `kill_thread` 方式直接粗暴杀死线程，这个有隐患，不建议采用，还有一种就非常简单了，既然线程是函数，当函数返回时，线程自然终止。

本书介绍的方法，全部是采用线程自然终止方式，温和退出线程。

那这里就带出来一个问题，各个操作系统定义的线程回调函数构型不同怎么办？这也是我们在跨平台开发多线程程序必须要处理的一个难点。

经过查找资料学习，我们知道，一个标准的 Win32API 规定的线程函数构型如下：

```
unsigned int __stdcall ThreadFuncName(void* pParam);
```

大家注意到没有，操作系统为这个回调函数也透传一根 `void*` 指针，这是协助应用程序传递参数。该函数返回一个无符号整数，这个整数对 Windows 操作系统而言，没有意义，主要供应用程序自己做一些状态识别。

而 Linux 系统下，一个标准的线程构型如下：

```
void* ThreadFuncName (void* pParam);
```

这里透传的参数指针就不再赘述，但最大的区别，Linux 下执行的 Posix 线程规约，规定线程必须返回一个无符号指针，这大约是希望指向一组返回值的结构体指针，在本书中不使用这个返回值。

由于上述差异性导致跨平台开发的不便，因此，笔者这里给出一个简单的线程函数通用构型定义如下：

```
#ifndef WIN32
    #define THREADFUNCDECL(func,args) unsigned __stdcall func(PVOID args)
#else // not WIN32 (Linux)
    #define THREADFUNCDECL(func,args) void * func(void *args)
#endif
```

这样，我们可以在 C 程序中，很轻松地使用宏来定义一个线程函数。

```
THREADFUNCDECL(MyThread,pMyParam)
{
    //...
}
```

但是，这里有必要特殊说明的是在 C++ 类中的声明和定义。类中声明实例如下：

```
class CMyMoudle
{
private:
    //定义一个名字为 MyThread 的静态线程函数，其参数为 pParam
    static THREADFUNCDECL(MyThread,pParam);
};
```

而在线程函数实现时，由于 C++ 的类成员函数实现，需使用“类名+::+函数名”的形式标定命名空间，因此，其实例如下：

```
//线程函数实现，注意其函数名为 CMyMoudle::MyThread
THREADFUNCDECL(CMyMoudle::MyThread,pParam)
{
    //...
#ifdef WIN32
    return 0; //Windows 下返回整数
#else // not WIN32
    return null; //Linux 下返回空指针
#endif
}
```

以上过程，大家也可以看做函数型宏定义的一个使用实例。请注意，这类构型上的定义统一，只能使用函数型宏来完成。很多时候，笔者在讨论函数型宏时，一些朋友总是以 C++ 禁止使用宏，来批评这个话题，但是，函数型宏在 C++ 内部自始至终都没有被禁止过，因为 inline，在某些时候，确实没有办法完全替代函数型宏。

9.2.3 线程函数启动

既然不同操作系统的线程函数构型差异都这么大，其启动函数差别自然也很大，需要做一个统一的定义。

```
//请注意，Linux 版本下，id 这个参量其实没有使用
#ifdef WIN32
    #define THREADCREATE(func, args, thread, id) \
        thread = (HANDLE)_beginthreadex(NULL, 0, func, (PVOID)args, 0, &id);
#else // not WIN32 (Linux)
    #define THREADCREATE(func, args, thread, id) \
        pthread_create(&thread, NULL, func, args);
#endif
```

请注意，在 Windows 下，使用的是 `_beginthreadex` 这个函数，当启动成功后，返回线程句柄和线程 ID（注意 windows 下声明中 `&id`，这是传址操作）。

而不管 Windows 还是 Linux，最终，我们都可以获得有效的线程句柄。当然，由于前面说的原因，我们会抛弃这两个值，不做保存。

一般说来，我们启动一个线程，会按照如下方式调用。

```
THREADID id;           //定义线程 ID 变量
THREAD t;              //定义线程句柄变量
THREADCREATE(
    MyThread,           //线程回调函数名
    pParam,             //线程回调函数参数指针
    t,                  //线程句柄（获得）
    id);                //线程 ID（获得）
```

9.2.4 MIN_SLEEP

在讲述线程相关知识的时候，有一个概念，虽然和操作系统级的线程操作无关，但是与我们的线程函数编写却密切相关。这里笔者也一并列出说明。这就是睡眠（Sleep）。我们先看看声明定义

```
#ifdef WIN32
    #define MIN_SLEEP 10           //Windows 下最小睡眠精度
#else // not WIN32
    #define Sleep(ms) usleep(ms*1000) //为 Linux 创建的 Sleep 调用
    #define MIN_SLEEP 1           //Linux 下最小睡眠精度
#endif
```

这里面有两点需要特别说明：

1、Linux 下没有 Windows 下的 Sleep 函数，但是，为了保证我们跨平台程序开发的统一性，我们使用 Linux 下精度更高的 `usleep` 函数，以函数型宏定义方式，定义了一个 Sleep 的调用宏。

2、Windows 的精度没有 Linux 高，Windows 下最小睡眠精度，经实测大约是 10ms 左右，即 `Sleep(5)` 和 `Sleep(10)`，实际睡眠时间是差不多的，而 Linux 下明显高得多，精度可以达到 1ms，甚至更高。因此，`MIN_SLEEP` 这个宏就变得很有必要，同样的服务器代码，Linux 下运行效率远高于 Windows 下，这里就是很大一个原因。

9.2.5 线程操作总结

最后，笔者给出一个跨平台开发线程的基本模型。请各位读者参考。我们以一个 C++ 的类来说明，以尽量模拟最复杂的调用情况。

在这个类中，我们将在构造函数中启动 100 根线程，并在析构函数中全部安全关闭，线程控制方法就是使用前文多线程安全的变量模板中，我们提供的管理方式。即以一个 `bThreadContinue` 和一个 `nThreadCount` 两个线程安全变量来管理。

这段代码虽然仅为示例，但是确实是可以运行的，各位读者有兴趣的话，可以在线程中添加一定的打印语句，实际编译运行。

9.2.5.1 类声明

首先，我们看看类声明，请注意，本类使用了前文所述多线程安全的变量。

```
//跨平台线程开发演示类
class COSThreadTest
{
public:
    COSThreadTest();           //构造函数
    ~COSThreadTest();          //析构函数
private:
    static THREADFUNCDECL(MyThread, pParam); //静态线程回调函数声明
    CMint m_nThreadCount;        //线程计数器
    CMbool m_bThreadContinue;    //线程可持续运行标志
};
```

9.2.5.2 构造函数

在构造函数中，我们将启动 100 根线程，注意，是间隔 250ms 启动，避免操作系统的禁忌。

```
COSThreadTest::COSThreadTest()
{
    m_bThreadContinue.Set(true); //初始化变量，线程持续标志置为真
    m_nThreadCount.Set(0);       //线程计数器清空
    int i=0;                     //中间循环变量
    THREADID id;                 //注意，这就是线程相关的两个变量
    THREAD t;
    for(i=0;i<100;i++)           //循环 100 次，启动 100 根线程
    {
        m_nThreadCount.Add();    //启动前先累加线程计数器
        //启动线程，请注意，我们没有保存任务 id 和 t 的值，因为不使用。
        THREADCREATE(MyThread,   //这 100 根线程都是相同线程函数的实例
            this,                //以本对象指针作为参数传入
            t,                    //线程句柄
            id);                 //线程 id
        Sleep(250);              //强制睡眠 250ms 间隔，避免操作系统禁忌
    }
}
```

9.2.5.3 线程函数

在线程函数中，我们模拟常规的函数模型，即以一个主循环作为函数主要业务逻辑。

```

THREADFUNCDECL(COSThreadTest::MyThread,    //函数名
                pParam)                    //参数
{
    //注意此处，强制指针类型转换，获得本对象指针 pThis，
    //因为下文我们需要访问类成员变量 m_bThreadContinue 和 m_nThreadCount
    COSThreadTest* pThis=(COSThreadTest*)pParam;
    //初始化代码放在此处
    while(pThis->m_bThreadContinue.Get())    //循环检测线程持续标志，为假退出
    {
        //...程序逻辑代码
        Sleep(MIN_SLEEP);                  //必不可少的 Sleep，时间片规避
    }
    pThis->m_nThreadCount.Dec();              //退出前，线程计数器-1，供主线程检测
#ifdef WIN32
    return 0;                              //Windows 下返回整数
#else // not WIN32
    return null;                            //Linux 下返回空指针
#endif
}

```

9.2.5.4 析构函数

析构函数中，我们安全关闭所有线程，再行退出。

```

COSThreadTest::~COSThreadTest()
{
    //线程持续标志置为假，这样，每个线程看到会自行退出
    m_bThreadContinue.Set(false);
    //不断检测线程计数器的值，当为 0 时，所有线程已经安全退出
    while(m_nThreadCount.Get())
    {
        Sleep(MIN_SLEEP);    //主线程也是线程，循环内也需要 Sleep
    }
    //此时，所有线程一定已经安全退出
}

```

9.2.5.5 线程调用实例小结

本例的模型，已经基本覆盖了跨平台线程安全控制的方方面面，在很长一段时间内，笔者在商用并行工程中，都是这么书写线程调用，从来没有出过问题，各位读者可以参考。

大家可以看到，线程的安全调用，需要关注到的细节还是很多的，稍有不慎，就可能造成 bug，因此，笔者在后文将向大家介绍线程池和任务池，当这两个工具完成，线程的安全调用已经可以使用更加简便的方法，无需这么复杂。

9.2 线程池

多任务操作系统，原则上其执行生命体，包含进程和线程两种形式。在 Windows 下，进程更像一个主线程，本质还是以线程方式运行，这说明，Windows 下，线程是执行生命体的实例样本。而 Linux 下恰恰相反，由于 Unix 的渊源，最开始的 Linux 下并没有线程，但是有子进程的概念，所有的执行体都是进程格式，Linux 现在的线程，更像是子进程的伪装，因此，在 Linux 下，进程是执行生命体的具体实例。

但是，这些对于应用程序员来说，并不重要，二者执行的机制虽有差异，但二者开发的原则如出一辙，都是强调变量的私有，基于锁的安全访问，以及礼貌地规避等。在本书中，笔者把这类同一进程内的执行生命体，统一抽象称之为“线程”。

我们知道，不管是进程还是线程，隶属于同一个应用工程的各个执行生命体，总的来说还是需要发生联系的，要通过彼此互动来完成有意义的工作，不可能绝对的没有任何关系，那就不是一个应用程序了。

在商用数据传输工程中，进程通常被用来区别网络角色定位的基本单元，为了保证网络角色的最大部署灵活性，一般进程间通信，都建议采用 `Socket` 传输，尽量避免操作系统建议的共享内存区、临界区、信号量等形式，以避免人为绑定两个应用服务进程，必须运行于同一台计算机。

因此，本书中讨论的多任务计算，多个执行生命体之间的关系，更多得是基于线程来讨论，因为变量的争用，多任务的协调，更多的出现在线程之间。这也是为什么我们构建线程池的原因。

9.2.1 线程池的来源和需求分析

笔者之所以开发线程池，最原始的动机，主要是为了管控线程的起停行为，实现安全的线程起停。根据经验，笔者认为，在线程工作时，至少有几点原则是必须关注的：

1、前文我们讲过，线程不能太密集地启动，最好前一线程和后一线程的启动，间隔 250ms 以上，方能安全。

2、线程永远不允许在外部关闭，否则，线程内动态申请的资源，很可能因为得不到释放而导致泄漏。

3、线程是执行生命体，线程之间的关系，是平等的，包括主线程在内，都不拥有对其他线程的生杀大权，只能使用温和协商的方式启动和关闭线程。

而这些原则如果应用到工程之中，如果分散管理，就会带来很多麻烦，甚至很多 bug，因此，使用线程池，收拢所有线程相关的逻辑操作，对外提供统一的界面，就变得很有必要，可以以此简化开发工作，减少 bug 的可能性。

9.2.1.1 线程的抽象理解

前文我们说过，线程如果抽象出来，就是操作系统根据应用程序的注册请求，为应用程序实现的回调服务，线程函数，是操作系统级的回调函数。从这点说，线程和进程是没有什么差别的。

如果我们把内存中一段正在执行的程序，视为一个有生命的物体的话，每个线程都是一个独立的执行生命体，从某种意义上讲，它是“活”的。即，一条线程，是有自己独立生命（执行时间片），有自己的独立思想（程序逻辑）的生命体，是可以主动和其他线程交互，并主动访问某些资源的执行逻辑。

笔者认为，我们可以从两个方面来看待线程：

1、线程的执行，从物理上看，是片段的，是断续的。从执行机制上看，线程天生就是被分割成片段在执行，它的执行总是“执行---被打断---再被执行---再被打断---...”这样一个过程。因此，线程的设计者，不能默认自己的程序是连续被执行的。任何一个函数，任何一条语句，甚至任何一个表达式，都可能在执行中被打断。

这说明，线程仅仅是资源的借用者，而不是资源的拥有者。以内存为例，一个进程内部的所有线程，共享该进程的私有内存空间，一个全局变量，一定可以被所有线程合法地访问。任何线程在访问这些资源的时候，都有义务实现一些宣告类的工作，通知其他兄弟线程进行规避，这就是锁的作用。

但是，即使我们已经小心地使用锁，进行了安全的线程设计，仍然远远不够。我们必须深刻理解，时间片是多任务系统最重要的资源，任何一个线程，都必须小心地使用时间片，随时注意关注自己的业务逻辑，是否可以临时告一段落，及时出让时间片给其他兄弟线程，做到礼貌地运行。

2、从逻辑抽象上看，线程又和普通程序并无不同。由于操作系统和编译器为每个线程开辟了独立的浮动运行栈，线程函数的执行，子函数的调用，各个业务逻辑的实现，又具有相对的独立性，与其他兄弟线程没有太多的干涉。

从线程自身的角度看来，它在运行时，仿佛是系统资源的独立拥有者，只要按照一定的原则，谨慎地访问各种资源，线程逻辑完全可以像其他单任务系统的程序一样，自如地运行。

笔者的结论：这种**物理上的片段性**，以及**逻辑上的连续性**，就是线程运行模型的核心特点。

9.2.2.2 任务的抽象理解

在讨论线程池之前，我们有必要先讨论一下什么叫做一个操作系统的“任务”。

简单的讲，任务就是一段应用程序，它能被操作系统以一定规则进行执行。既然任务是一段程序，那么我们需要先理解一下计算机程序的抽象模型。这很重要，因为后续我们如何设计我们的回调函数，需要用到这部分知识。

相信所有本书的读者，或多或少，都写过一些程序，那么，一段计算机程序，究竟有哪些抽象特征呢？

首先，我们确定一点，一段计算机程序，应该包含一段计算逻辑，即根据一些数据，以一定的计算规则，得到另外一些结果数据。

在此之前，程序应该有一些必要的初始化代码，以便为后续的计算申请资源，准备数据，在计算结束之后，一般说来，也应该有一些结束代码，以便释放资源，达到安全运行的目的。如下图所示：

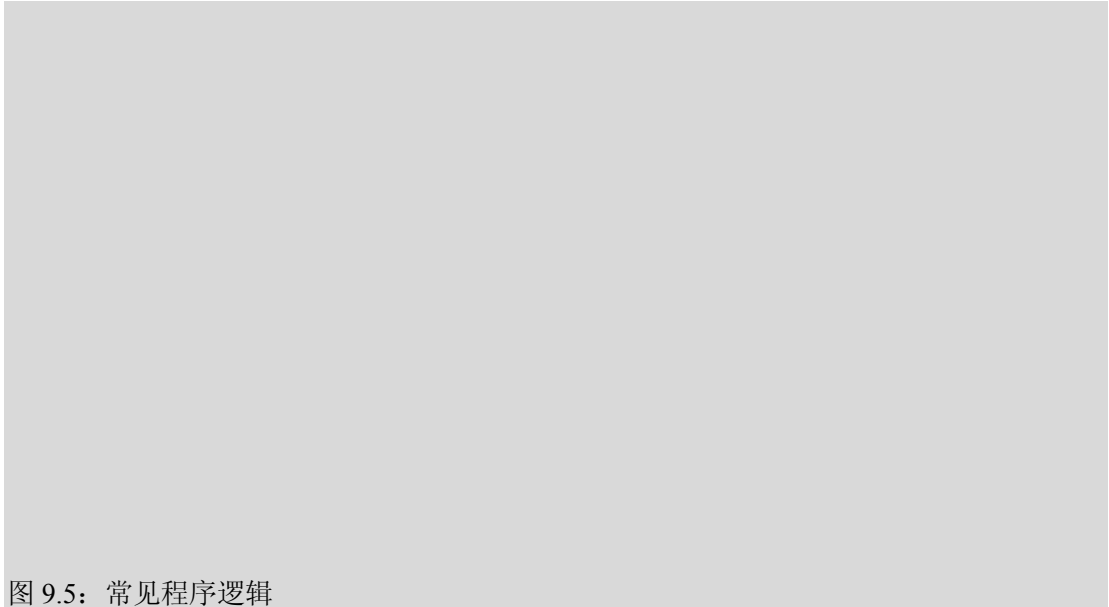


图 9.5: 常见程序逻辑

但是，如果我们多开发一些代码之后，通常会发现，其实上述模型还可以进一步细化。虽然没有任何规定，以及原理论证，但是，我们可能会发现，通常上述程序中间的执行体，只要计算稍微复杂一点，就是一个循环体。这完全是经验之谈，没有什么理论依据的，那



图 9.6: 循环式程序模型

么，整个程序的模型，就变成了如下形状：

因此，如果我们要抽象面向过程型的程序流程的普遍特征，一般可以论证，一段完整意义的代码，一个函数，一般说来由三部分构成：

- 1、初始化代码
- 2、一个条件循环
- 3、结束代码

我们可以想象，任何一段程序，无论大小，基本上都可以以上述模型进行描述。当然，也有部分代码，中间执行简单计算，没有那个循环过程，顺序执行完毕即退出，不过，这毕竟是少数。

9.2.2.3 线程的管理需求

我们在 C 和 C++ 的无错化程序设计方法中提到过，每段逻辑只写一次。既然我们知道了线程的运行特性，以及线程开发的原则、禁忌，那么，我们首先同意，线程的管理，也是一段逻辑，如果我们在程序中多次书写这段逻辑代码，会造成程序冗长，不够简练，同时也很容易出现 bug。

那么，我们是不是可以考虑，以某种程序模块的形式，来封装线程管控的一些核心逻辑，屏蔽线程管理的细节，让后续的应用程序，专心于自己的业务逻辑，不再随时关注线程开发的限制，更好地完成功能呢？答案是肯定的。这也就是笔者开发线程池的由来。

从另外一个方面，我们也能看到开发线程池的必要性。我们知道，线程启动需要一定的控制，不能太频繁，但在实际业务中，我们很多时候很难精确控制线程启动的频率。

举个例子讲，商用数据传输系统的服务器设计，通常内部设置 Listen 线程，专门监听某一端口，响应客户的连接请求。这样的话，当一个客户请求到来时，Listen 的常见做法，是 accept 一个 socket，然后立即开辟一个线程，开始处理这个 socket 的传输行为，而自身则返回来继续监听，准备处理下一连接请求，这是保证高吞吐量的关键。

但是，这就带来一个问题，公网运行的服务器，其客户连接请求是并发的，频繁的，也是不可预知的，相邻的两个连接请求，可能之间连 1ms 的差距都没有，如果我们使用上述逻辑开发服务器，则必然无法控制线程启动的频率，此时，使用线程池实现管控，调整线程启动的步距，就变得非常必要。

还有一个原因也是我们需要使用线程池的理由，我们知道，线程的启动，是一种操作系统的注册行为，需要在系统的核心任务机链表中，填充一些信息。而此时，出于安全访问的原因，系统的任务机，一定使用了某种锁保护策略，这个锁，不仅仅是线程安全锁，而是进程级安全的。因此，锁使用的成本很高。

如果一个进程无原则的随意乱起线程，很可能造成核心任务机的注册行为太过于频繁，我们知道，操作系统分配时间片，本身也需要遍历这个链表，读取数据，以便实现回调。而注册行为太频繁，会对整个操作系统的执行效率造成严重冲击，使正常的时间片分配流程，频繁地因为注册动作而陷入长时间等待，这种粗暴地冲击行为，往往造成整个操作系统的运行不稳定，应用程序应该予以规避。

因此，我们总结一下，如果要设计一个线程池，关键需要实现如下需求：

- 1、线程池作为一个进程的唯一线程管理单元存在，负责所有线程的启动和停止工作。
- 2、线程池维护线程的安全启动工作，即每两个线程启动的时间间隔大于 250ms。
- 3、线程池负责线程的安全退出工作，当应用程序需要退出，中断所有的活动线程时，线程池应该提供方法，通知所有线程温和地退出，释放所有资源。
- 4、除此之外，线程池还应该负责平衡线程频繁起停对操作系统的冲击，使多线程开发的应用程序，能以较为温和的方式，和操作系统实现交互，避免太过于密集的注册风暴，冲击操作系统的运行效率。

9.2.2 线程池的设计原理

从本小节前面部分，我们已经得知，多任务程序员，有必要设计一个线程池来缓和业务需求和操作系统之间的摩擦，实现温和的，安全的线程管控，那么，我们有必要讨论一下，究竟怎样才能实现这一目的。即：线程池究竟怎么做？

这个道理说起来比较复杂，笔者也经过了较长时间的思考，但一旦想通，其实非常简单。

9.2.2.1 线程池基本设计

我们知道，线程实际上是操作系统级的回调函数，我们启动一根线程，其实是想操作系统任务机注册一个任务，操作系统就会授予我们执行权，在将来某个时刻，调用我们的代码，实现程序业务逻辑的执行。如下图。



图 9.7: 无线程池，操作系统直接回调线程

那么，我们可以把思路反过来想一下，我们启动一个线程，其实就是向操作系统“索要”了一组时间片来执行任务，但是，操作系统并没有界定我们执行什么任务，我们当然也可以建立一个自己的回调机制，通过回调业务层逻辑，把这个执行权有选择地，动态地授予需要的应用模块。

这实际上是在操作系统层和我们的应用层之间，再添加了一层回调逻辑，操作系统回调我们的线程函数，我们再根据业务层的注册情况，回调业务层逻辑，通过这种中间插入

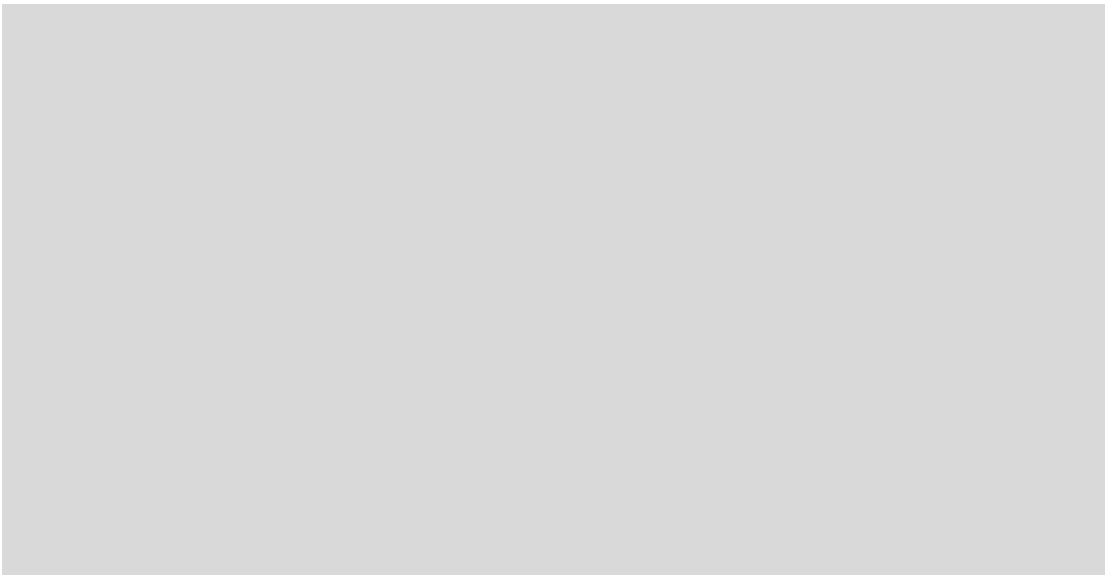


图 9.8: 有线程池，操作系统先回调线程池内线程，再由线程池二次回调到真实的业务线程一层的形式，实现线程池的管控逻辑。如下图。

想通了这点，我们基本上可以实现出构想中的线程池：

1、我们设计一个数据结构，链表或者数组都可以，因为我们预设处理一个应用进程内部的有限线程请求。

2、线程池模块提供注册机制，允许应用层注册新的线程任务到我们的线程池中。

3、线程池根据注册情况，以 250ms 为间隔步距，温和启动线程。

4、线程池提供 Close 方法，一旦应用层调用，即宣告系统即将退出，以某种方式温和关闭所有的线程，安全退出。

5、线程池一旦启动，除了必要的业务线程之外，通常预开设不大于 10 条（经验参数）线程的预备线程，这些线程平时空转，遇到注册任务，优先从这些预备线程中选择一条执行任务，一次缓和频繁的起停冲击。当预备线程不足 10 条时，线程池继续开启预备线程，始终保持预备线程的数量在 10 条左右。

6、当有业务线程退出，预备线程数超过 10 条，多余的预备线程自动退出，以保持预备线程数，始终在 10 条左右。

7、由上得知，线程池内的线程，具有两种状态，正在执行任务的 Busy 状态，以及作为预备线程，等待任务执行的 Idle 状态。

8、线程池本身的所有业务逻辑，线程安全。

9.2.2.2 线程池动态温和启动管理原理

我们如果要实现线程的安全启动，首先就必须摒弃传统的 `begin_thread` 的做法，虽然各种系统都以这个函数，作为启动线程的方法，但是，我们要清楚地认识到，这个函数本身是同步调用，即应用程序一旦调用，线程立即启动，这无法实现 250ms 的间隔启动步距，以一种较慢速和温和的方式启动线程。

这就要求，线程池必须建立一套机制，把原来同步的线程启动动作，变为异步的，即应用层启动新线程的动作，和线程真实启动的动作无关。

因此，笔者首先考虑独立启动一个线程池的管理线程，这个线程属于守候线程，不断循环监控目前的预备线程数量，如果低于 10 条，则以 250ms 的步距，启动新线程，直到数量达到 10 条要求。

另外，启动的预备线程，则以守候的方式检测一个共享变量区，如果发现新的任务注入，则回调该任务，实现应用层线程的真实执行。当预备线程无任务时，检测预备线程数量是否超过 10 条，如果是，则退出自身，以维护预备线程总数在 10 条左右。

新的应用层线程启动动作，现在变成了一个注册方法，首先，其检测目前是否有空闲的预备线程，如果没有，就返回或者等待，当检索到空闲的预备线程，则将任务注册到该预备线程的任务区，由预备线程开始执行，此时返回成功。

这样，应用层线程启动函数，线程池管理线程，预备线程三者各自独立运行，互不干涉，形成异步工作模型。当然，此时的共享变量区必须做好锁保护，以保证异步安全。

9.2.2.3 线程安全退出原理

在前文我们说过，线程的安全退出，可以考虑使用两个变量来实现，即 `bThreadContinue` 和 `nThreadCount`。二者的职能划分如下：

1、`bThreadContinue` 是全局所有线程能持续运行的标志，该值为真时，所有的线程循环正常运行。一旦被设置为假，则所有的线程跳出循环，执行退出代码，结束运行。这个变量，体现了主线程对各个子线程的管控，以此来控制所有子线程的生死。

2、`nThreadCount` 这个变量可以视为是子线程对主线程的反馈，每个子线程启动，会将其+1，退出会将其-1，主线程将 `bThreadContinue` 设置为假之后，必须要等待一段时间，等待所有的子线程退出，方能安全退出，否则，易引起崩溃。

由上得知，我们可以使用这两个变量，实现主线程和子线程之间完美的互动，最终实现线程安全退出的机制。

线程池内置上述两个变量，当然，这两个变量由于被多个线程同时读写，因此，一定要采用前文介绍的多线程安全变量来实现，以保证其自身逻辑的安全性。

9.2.2.4 线程池基本拓扑模型

经过一轮思考，我们现在可以简要画出线程池的基本拓扑模型：



图 9.9：线程池拓扑模型

大家可以仔细看看这幅图，在整个线程池中，有四个关键角色：

- 1、线程池管理线程
- 2、任务执行线程（真实线程）
- 3、任务执行线程区
- 4、应用线程注册方法（主线程或其他线程）

这中间的关键，请大家关注，在任务执行线程区内，有多个线程任务区，这是一个结构体数据块，这也是几个角色交互的中心，每个真实执行线程，拥有唯一一个任务区，但每个任务区，**不一定**有真实的执行线程为其服务。

线程池管理线程，不断循环检索任务区所有的任务块，一旦发现空闲的真实执行线程少于 10 条，即启动真实执行线程，总之保证 Idle 的预备线程数在 10 条左右。

应用线程注册方法则检索任务区的任务块，找到第一块有活动的执行线程，并且处于 Idle 状态的任务块，将任务注册进去。

而活动的执行线程，不断检索自己的任务块，发现任务，就执行，返回后，清空任务块，等待下一次任务的到来。当长期没有任务，出于 Idle 状态时，检查当前 Idle 的线程是否超过 10 条，超过则退出自己，以保证预备线程数在 10 条左右。

以上即为笔者设计的线程池基本工作逻辑。

9.2.2.5 线程回调函数基本构型

前文我们讲述了一个任务的基本形态，我们基本可以知道，一段普遍意义上的程序流程，包括初始化代码，一段循环代码，以及结束代码三部分。

这里就带出来一个问题，我们知道，线程池的真实系统线程，一旦开始回调注册任务，就把执行权交付给应用程序线程模块，除非这个模块自行返回，否则线程池是没有办法收回执行权的。

因此，笔者设计的线程回调函数构型，最关键的是把 `bThreadContinue` 这个参数传递进去，这样，当主线程需要退出时，应用层线程也能看到这个值的变化，以便退出任务。其构型如下：

```
//线程池回调函数指针
//每个回调函数有一次运行权
//运行结束,线程并不退出,进入 IDLE 状态
typedef void (*_TPOOL_CALLBACK) (
    void* pCallParam,          //这是主程序传进来的参数指针
    MBOOL& bThreadContinue);    //如果系统要退出,线程池会修改这个参数的值为 false
//习惯写法,写出一个回调函数构型,立即给出示例,方便以后的应用程序拷贝使用
//static void ThreadPoolCallback(void* pCallParam,MBOOL& bThreadContinue);
```

笔者为线程池回调函数，设计了两个参数，一个就是 `bThreadContinue`，另外一个，是基于回调函数的特点，为应用层透传的一根 `void*` 指针，帮助应用层调用者传参。

这样，我们可以看看一个普通的应用层回调函数的基本模型和关键元素：

```
void AppThread(void* pCallParam,MBOOL& bThreadContinue)
{
    //初始化代码
    while(XMG(bThreadContinue)) //主业务循环,必须参考传入的 bThreadContinue 值
    {
        //业务代码...
        Sleep(MIN_SLEEP);      //必要的最小睡眠
    }
    //结束代码
}
```

一般说来，按照前述的程序三部分理论，我们可以预设应用层线程如本例所示，也分为三段。最关键的是中间的 `while` 语句，不断判断传入的多线程安全变量 `bThreadContinue` 是否为真，以此作为循环可以持续的标志。

当然，如果一个业务线程逻辑够简单，不需要循环，则取消中间的 `while` 判断即可。

提示：根据笔者的经验，一个应用层线程只有按照上述示例书写代码，方能保证线程池安全可靠运行。

9.2.3 线程池的基本数据结构

在开发真正的线程池之前，我们有必要先预设一下线程池需要处理的数据结构。

9.2.3.1 各种常量设计

一般说来，线程池是应用程序员开发的，专门为应用程序服务的模块，因此，常规的规模控制，都是由一些宏常量来管理，以便随时根据业务需求更改。线程池主要的常量，有以下几条。


```

#define OPEN_THREAD_DELAY 250 //新开线程的延迟 ms 数,Windows 和 Linux 系统,
//开线程太快的话,会出现死线程
//必须有一个延迟,经验参数建议>200ms
#define WHILE_THREAD_COUNT 10 //最多 n 条线程空闲等待任务 (经验参数: 10)
//这不是硬性的,如果有线程已经 IDLE,
//可能会比这个多,但随后会动态调整
#define DEFAULT_THREAD_SLEEP 500 //通常建议的线程睡眠参数

#define THREAD_POOL_EXIT_CODE 10000 //线程池的线程,从开始设置退出码
//总线线程数上限
#ifdef _ARM_ //ARM 嵌入式系统
#define THIS_POOLTHREAD_MAX 30 //嵌入式系统下最大 30 条线程
#else //PC 机
#ifdef WIN32
#define THIS_POOLTHREAD_MAX 2000 //Windows 下最大 2000 条线程
#else // not WIN32
#define THIS_POOLTHREAD_MAX 300 //Linux 下最大 300 条线程
#endif //WIN32
#endif
#endif

```

上述常量,大家应该不难理解,OPEN_THREAD_DELAY,这就是我们前文说的,线程开启时,250ms 的时间间隔常量。WHILE_THREAD_COUNT,这是处于 Idle 状态的预备线程数量,一般为 10 条,这是经验参数,可以根据实际情况调整。

DEFAULT_THREAD_SLEEP,这是 Idle 线程的睡眠基数,一般为 500ms,即每 500ms,线程会苏醒,检测是否有新任务到来。这个数字一般不宜太小,否则 Idle 线程启动太频繁,会影响系统效率。

可能最难以理解的,就是 THIS_POOLTHREAD_MAX 这个常量了,最大线程数。我们知道,任何一个系统都不可能是无边界的,因此,线程池需要设置一个上限,不能无原则地增加线程,否则最终会把系统挂死。

在 Windows 下,原则上不限制线程数的,不过,笔者这么多年的经验,一般单 CPU 下,2000 条应该是极限了,不过,一般都用不到这么多。

Linux 下的情况稍稍特殊一点,因为 Linux 其实还是鼓励大家使用子进程的,线程其实也是子进程伪装的,因此,一般的 Linux 内核,如果不做特殊编译的话,每个进程内线程数一般限制在 310 条左右,再多,就开不出来了。这里笔者减小一点点,建议 300 条。

本书预设的嵌入式平台样板是基于 arm9 的 Linux 系统,这个情况最为特殊。我们知道,线程是需要独立浮动栈空间的,嵌入式 Linux 下,一个线程预设栈空间,一般是 1M。而 arm9 的开发板,一般标配 64M 内存,这说明,最多开 64 条线程。

通常还没有这么多,笔者请教过一些专业开发嵌入式系统的朋友,也自己做了一点试验,基本上,50 条线程是极限,因为毕竟 Linux 内核也有线程,也需要占据内存。

这里,笔者建议 30 条左右,主要是考虑还要为堆空间保留一点内存。这里也可以看出来,嵌入式系统的开发环境比较恶劣,资源很少,程序员必须小心谨慎地使用内存。

另外,这里还有一些线程池的状态定义。

前面我们说过,线程池内的线程,一般有两态,一个是有任务的 Busy,一个是空闲的 Idle,但是,我们参考一下图 9.8,我们可能发现,线程池内的任务块,并不是每个任务块都具有激活的线程为其服务,因此,它的状态实际上是三态,还有一态是没有线程服务,TPool_Thread_State_Not_Run。

```
#define TPOOL_THREAD_STATE_NOT_RUN 0    //线程池线程状态,线程未运行
#define TPOOL_THREAD_STATE_IDLE 1      //线程池线程状态,线程运行,空闲
#define TPOOL_THREAD_STATE_BUSY 2      //线程池线程状态,线程运行,有任务
```

9.2.3.2 线程池基本管理数据结构

```
class CTonyThreadPool;
typedef struct _THREAD_TOKEN_
{
    int                m_nExitCode;        //返回值,也是本线程在整个线程池的编号
    MINT               m_nState;          //线程状态机
    THREAD             m_hThread;         //线程句柄
    THREADID           m_nThreadID;       //线程 ID
    _TPOOL_CALLBACK    m_pCallback;       //回调函数
    void*              m_pCallParam;      //回调函数参数
    CTonyThreadPool*   m_pThreadPoolObject; //指向线程池对象的指针
}SThreadToken;
const unsigned long SThreadTokenSize=sizeof(SThreadToken); //结构体长度常量
```

线程池的基本数据管理结构比较复杂,这里先给大家建立一个感性认识,后面的程序段落中,我们再详细了解。

这里可能大家比较费解的是结构体声明前的线程池类声明“class CTonyThreadPool;”。这是 C++ 书写代码一种常见的手法,这里给大家介绍一下。我们看一段程序实例:

```
class CTest; //使用预声明
typedef struct _TEST_STRUCT_ //数据结构体
{
    CTest* m_pFather; //内部包含父类指针
}STestStruct;
class CTest //父类
{
    STestStruct m_Array[100]; //内部包含结构体数组
};
```

请大家仔细看,STestStruct 作为 CTest 类使用的一个数据结构体存在,但是,其内部由于业务需要,包含了一根 CTest 指针,我们可以站在编译器的角度看待这段代码,当从前向后顺序编译时,编译器扫描到 STestStruct 声明时,CTest 还未被定义,因此会引发一个标签未定义的错误,编译无法进行。

解决方法很简单,就是在结构体之前预先声明一下 CTest 的存在,编译器就知道 CTest 是一个类,至于这是什么类,下面有说明,因此就不再报错。这个技巧很常见,尤其是在 MFC 里面,由于各个类库之间错综复杂的依赖关系,很多时候,会发生这类交叉现象,都需要做这种预声明。

9.2.4 线程池的类设计说明

线程池不同于内存池。内存池是“被动池”,即只提供资源,不提供执行手段,永远被动地等待其他执行生命体来调用。而线程池是典型的“主动池”,即对外提供执行生命体资源,永远是动作发起的主动方。因此,线程池的设计,相对比较复杂。

9.2.4.1 线程池类声明

线程池虽然比较复杂,但由于其仅提供几个简单的注册方法函数,因此类声明还是比较简单的。但是,其内部包含的意义非常深刻,这在下面的小节有专门论述,请各位读者仔细体会。

```

//线程池类
class CTonyThreadPool
{
public:
    CTonyThreadPool(CTonyLowDebug* pDebug);           //需要传入 Debug 对象指针
    ~CTonyThreadPool();

public:
    //注册一个新线程，返回状态值，状态值后文有叙述
    int ThreadPoolRegTask(
        _TPOOL_CALLBACK pCallback,                    //回调函数指针
        void* pParam,                                  //代传的参数指针
        bool bWait4Success=TRUE);                     //是否等待注册成功才返回
    bool TPAllThreadIsIdle(void);                     //检查所有线程是否空闲
    bool ThreadPoolIsContinue(void);                  //检查线程池运行状态

private:
    //这里是真实的操作系统线程函数，其构型后文叙述
    static THREADFUNCDECL(ThreadPoolThread,pParam);   //线程池服务线程
    static THREADFUNCDECL(ThreadPoolCtrlThread,pParam); //线程池管理线程

private:
    //内部函数，检索没有使用的 Token
    int Search4NotUseToken(void);
    //内部函数，获得一个空闲线程
    int GetAIdleThread(void);
    //这是完成具体注册动作的内部函数，后文详细叙述
    int ThreadPoolRegisterANewThread(
        _TPOOL_CALLBACK pCallback,
        void* pParam);
    int ThreadPoolRegisterANewThreadWhile(
        _TPOOL_CALLBACK pCallback,
        void* pParam);

private:
    SThreadToken    m_TToken[THIS_POOLTHREAD_MAX]; //线程池任务参数静态数组
    //这两个参数就是线程池安全退出控制参数，int 型的计数器+bool 型的线程持续标志
    MBOOL           m_bThreadContinue;              //所有 Thread 继续标志
    MINT            m_nThreadPoolThreadCount;        //Thread 计数器
    //统计变量
    MINT            m_nThreadPoolIdleThreadCount;    //空闲的线程数量
    //线程池没有使用前述的 C++锁类，而是直接使用 C 的纯锁结构体完成
    MUTEX           m_RegisterLock;                  //线程注册临界区
    CTonyLowDebug*  m_pDebug;                        //Debug 对象指针
};

```

9.2.4.2 线程传参原理

线程池内部，当然包含线程。我们前文说过，线程传参是一个比较复杂的问题，必须使用远堆传参，即不遵守“谁申请，谁释放”原则，以动态申请的内存块，为线程传参。

但这种情况显然不适合线程池的工作逻辑，在线程池内部，实际上包含了很多个执行体，每个执行体，包含一个任务数据描述结构 SThreadToken，和一条正在工作的，真正的操作系统级线程。

我们对外提供线程服务，其实就是允许外部模块，注册一个线程回调函数到某一个执行体的任务数据区，而该任务数据区的线程，会在下次苏醒时，检索到这个任务，从而给予这个线程函数一次执行机会。如图 9.9



图 9.10: 线程池内一个执行体

如图，线程池为每一个注册进来的任务回调函数，提供有且仅有一次的执行机会，当然，如前文所示，该函数可以通过内部循环，以不返回形式获得长期执行效果。但对于线程池而言，每个线程回调函数只有一次执行机会，线程在执行完任务后，会清空数据区，等待下一次任务注入。

我们前文还说过，任务数据区和线程执行生命体是一一对应关系，但是，出于节约资源考虑，线程池内的任务数据区可能会很多，但不是每个任务数据区都包含一个激活的执行线程。执行线程的启动是线程池管理线程动态管理的。这说明，**任务数据区的生命，比执行线程要长。**

这很重要，因为这决定了线程池内的真实执行线程以何种方式传参。远堆传参我们知道，是执行线程负责释放，因此，其生命周期和执行线程是等长的，不符合本类的设计需求，必须改变。

因此，经过考虑，笔者在线程池内部内置了一个任务数据区数组，以此来管理整个线程池中所有执行体的任务参数区。

```
SThreadToken    m_TToken[THIS_POOLTHREAD_MAX]; //线程池参数静态数组
```

由于这个数组的元素，属于线程池类成员变量，与对象生命周期等同，因此，任何时候，线程池提供的注册方法，都可以访问这个数组，实现任务注入。

而线程池每个服务线程，不是随意启动，每次启动前，线程池管理逻辑会为其提供数组中，某一成员的指针，以后该线程就为该任务单元服务，检索其中的任务并执行。

举个例子讲：线程池启动了一个执行线程，为其提供参数指针 `m_TToken[0]`，该线程就会始终围绕 `m_TToken[0]` 服务，不断检索其中的任务并执行，而 `m_TToken[1]` 则由另外的执行线程服务。

这就是线程池使用类成员变量实现的固定传参。因此，这里修正一点：“**线程传参不一定使用远堆传参的，只要能保证传递的参数区内存，其生命周期超过线程执行生命体本身的寿命，即可用于传参。**”

9.2.4.3 管理者线程和服务者线程

在正式介绍线程池之前，我们还有必要区别两个概念，即“管理者”和“服务者”。

这里有一个细节非常重要，笔者和大家必须达成共识。我们知道，每个进程一旦启动，其实至少已经有一条线程已经开始运行，就是“主线程”。这在 Windows 下较为明确，Linux 下虽然没有这么显式定义，不过，道理也差不多。而一个应用程序，基本模块，如内存池，和线程池等初始化工作，一般说来，就是在主线程中完成的，也就是说，线程池的构造函数和析构函数，是在主线程中执行。

由于这个原因，我们发现，不能在构造函数中，启动服务线程，否则，这个启动过程就变成同步动作了，无法满足异步，250ms 间隔启动的需求。

因此，线程池需要专门安排一条线程，来为线程池本身服务。这就是“管理者”线程。那么，相对而言，为每个应用程序任务服务的线程池内其他线程，就叫“服务者”线程了。

管理者线程负责不断启动服务者线程，满足任务需求。而服务者线程则不断检查有无新任务注入，有的话，就执行，无则 Idle 状态睡眠。二者互通，共同完成线程池服务功能。

当然，这里面还有个细节值得一提，注册函数 `ThreadPoolRegTask` 属于哪个线程，笔者的回答是哪个线程都可以，由于线程池的注册动作本身，具有线程安全性，新的任务，可以在主线程中注册，也可以在某一个服务线程任务中注册，都可以，这完全是异步动作，完全根据用户应用程序的需求来。甚至可以是线程池本身的服务者线程调用。

9.2.5 构造函数和析构函数

线程池的构造函数，负责整个线程池内部成员变量的初始化动作。

9.2.5.1 构造函数

线程池构造函数，最重要的就是初始化 n 个线程任务单元，为后续线程服务提供帮助。同时，构造函数还有个很重要的功能，启动管理者线程，以后服务线程的开启工作，将由管理者线程负责，主线程执行到这里，就返回了。这个异步模式，是线程池的第一个开发关键。

```

CTonyThreadPool::CTonyThreadPool(CTonyLowDebug* pDebug)
{
    m_pDebug=pDebug;                //保存 Debug 对象指针
    TONY_DEBUG("CTonyThreadPool Start!\n"); //显示开启信息
    THREADID id;                    //注意，开启管理者线程的参变量
    THREAD t;                       //函数内变量，表示不保存
    //注意，这里面没有使用锁对象，而使用纯 c 的锁结构体完成。
    MUTEXINIT(&m_RegisterLock);      //初始化线程池总的锁对象
    XMI(m_bThreadContinue,true);     //初始化 bThreadContinue 变量
    XMI(m_nThreadPoolThreadCount,0);  //初始化 nThreadCount 变量
    XMI(m_nThreadPoolIdleThreadCount,0); //初始化空闲线程统计变量
    //初始化数组
    int i;
    for(i=0;i<THIS_POOLTHREAD_MAX;i++)
    {
        //虽然管理者线程没有保存线程句柄，但考虑到应用线程以后潜在的需求，
        //在线程的任务结构体中，还是保留了线程句柄和线程 ID
        m_TToken[i].m_hThread=0;
        m_TToken[i].m_nThreadID=0;
        //这主要是为了调试方便，为每条服务线程设置一个显式的退出码，Windows 下，
        //线程返回可以看到这个值
        m_TToken[i].m_nExitCode=THREAD_POOL_EXIT_CODE+i;
        //任务描述，回调函数指针和透传参数指针，注意，null 表示当前无任务，
        //服务线程即使启动，也是在 Idle 下空转。
        m_TToken[i].m_pCallback=null;
        m_TToken[i].m_pCallParam=null;
        //这是一个很重要的描述，线程池自己的指针，原因后述。
        m_TToken[i].m_pThreadPoolObject=this;
        //初始化线程任务单元状态变量，注意，多线程安全的变量，c 模式
        XMI(m_TToken[i].m_nState,TPOOL_THREAD_STATE_NOT_RUN);
    }
    id=0;                            //开启管理者线程
    t=0;
    THREADCREATE(ThreadPoolCtrlThread,this,t,id); //注意，管理者线程的参数是 this
    Sleep(OPEN_THREAD_DELAY);         //强制等待管理者线程开启
}

```

这其中，比较难以理解的，大概是 `m_pThreadPoolObject` 这个参量，笔者设置这个参量时，是这么考虑的。

1、服务者线程需要首先观察分配给自己的这个任务结构体的值，因此，该结构体指针必须传递进线程。

2、但是，服务者线程还需要参考线程池本身的控制变量，`bThreadContinue` 和 `nThreadCount`，因此，线程池的 `this` 指针也必须传递给服务者线程。

3、这就要求传递两根指针，而操作系统的线程回调函数，一般都只允许传递一根 `void*` 的参数指针进入。

4、每个线程只能看到 1 个任务结构体，线程池中有 `THIS_POOLTHREAD_MAX` 个任务结构体，都需要赋值。

4、经过优化考虑，笔者在每个任务结构体中设置了 `m_pThreadPoolObject` 变量，传递线程池 `this` 指针，虽然这样占用了一定内存空间(`THIS_POOLTHREAD_MAX` 个指针空间)，但简化了很多传参计算，避免了 bug。

9.2.5.2 析构函数

线程池的析构函数，最重要的工作有两个：

- 1、安全退出所有线程
- 2、摧毁前面设置的所有线程安全变量和锁，完成二元动作的第二步，避免资源泄漏。

```
CTonyThreadPool::~~CTonyThreadPool()
{
    //关闭线程，这个段落比较经典，前文已多次出现，此处不再赘述
    XMS(m_bThreadContinue, false);
    while(XMG(m_nThreadPoolThreadCount))
        {Sleep(MIN_SLEEP);} //等待关闭所有线程
    //摧毁所有线程参数模块的状态变量
    int i;
    for(i=0; i<THIS_POOLTHREAD_MAX; i++)
    {
        XME(m_TToken[i].m_nState); //只有这一个需要摧毁
    }
    XME(m_bThreadContinue); //摧毁各线程安全变量
    XME(m_nThreadPoolThreadCount);
    XME(m_nThreadPoolIdleThreadCount);
    MUTEXDESTROY(&m_RegisterLock); //摧毁基本锁
    TONY_DEBUG("CTonyThreadPool Stop!\n"); //显示退出信息
}
```

9.2.4 管理者线程

管理者线程是整个线程池的核心，它负责不断检查线程池的服务线程数量，并确定其中处于 Idle 状态的服务线程数量，如果这个值低于默认的 10 条，则自动以 250ms 的步距，启动新的服务线程。它负责维护服务线程的增加逻辑，以保证服务线程，总是够用。

由于这个启动服务线程的过程在管理者线程中运行，因此，相对主线程而言，是异步的，因次，线程池也就是异步的。

9.2.4.1 服务函数 Search4NotUseToken

在介绍管理者线程之前，我们需要先看一个服务函数 Search4NotUseToken，该函数统计目前没有线程为之服务的任务块。

前文我们说过，线程池首先内置 THIS_POOLTHREAD_MAX 个管理任务数据结构，以应对并发 THIS_POOLTHREAD_MAX 个服务线程的可能性，但是，实际运行时，为了节约资源，并没有实际启动 THIS_POOLTHREAD_MAX 这么多条线程，也就是说，任务数据块和实际服务线程不是一一对应关系。

这就带来一个问题，管理者线程不能无目的地乱启服务线程，必须找到一个空闲的，没有线程服务的任务数据块，才能为其启动一根服务线程服务，这需要一个查找工作，即从任务块数组 m_TToken 中，查找到第一个未用的任务数据块，并以其为基础，开启一根服务线程开始服务。本函数就是实现这种查找行为。

这其实是说，任务数据块的空闲，和服务线程的空闲不是一回事，任务数据块有三种状态，1，真正空闲，没有线程服务，2，Idle，有线程，但是没任务，3，Busy，有线程，有任务。而服务线程的 Idle 和 Busy 仅对应后两种情况。

此处查找的是属于第一种情况的任务数据块。

```

//寻找一个没有使用的 Token
//找到, 返回编号
//没有找到(全部用满了), 返回-1
int CTonyThreadPool::Search4NotUseToken(void)
{
    int i;
    for(i=0;i<THIS_POOLTHREAD_MAX;i++)    //遍历整个数组查找
    {
        if(TPOOL_THREAD_STATE_NOT_RUN==XMG(m_TToken[i].m_nState))
            return i;                    //找到, 返回 index
    }
    return -1;                            //找不到, 返回-1
}

```

9.2.4.2 管理者线程函数 ThreadPoolCtrlThread

管理者线程相对逻辑比较简单, 就是不断检查处于 Idle 的服务者线程是否有 10 条, 如果没有, 就启动一根补全。


```

//管理者线程，注意：这里的函数声明是跨平台的线程函数定义，使用宏完成
THREADFUNCDECL(CTonyThreadPool::ThreadPoolCtrlThread,pParam)
{
    CTonyThreadPool* pThis=(CTonyThreadPool*)pParam;    //老习惯，获得 this 指针
    //注意，此处增加线程计数器，一般说来，线程池的使用者总是长期使用
    //即管理者线程总有实例化运行的机会，因此，线程计数器在其中+1，而不是放在构造函数
    XMA(pThis->m_nThreadPoolThreadCount);
    int nIdleThread=0;                                //空闲线程计数
    int nNotRunThread=0;                                //未运行线程计数
    //请注意这个死循环，参考 bThreadContinue
    while(XMG(pThis->m_bThreadContinue))
    {
        //获得当前空闲的线程数
        nIdleThread=XMG(pThis->m_nThreadPoolIdleThreadCount);
        if(WHILE_THREAD_COUNT>nIdleThread)
        {
            //如果备用的空闲线程不够 10,需要添加，则启动新服务线程
            //启动前，需要先找到空闲的任务块，找不到，也不启动。
            nNotRunThread=pThis->Search4NotUseToken();
            if(-1!=nNotRunThread)
            {
                //启动线程
                THREADCREATE(ThreadPoolThread,            //服务者线程名
                    //注意，此处把任务数据块指针作为参数传给服务者线程，
                    //因此，每个服务者线程，仅能维护一个任务数据块
                    &(pThis->m_TToken[nNotRunThread]),
                    //注意，此处保存了服务者线程的句柄和 ID
                    pThis->m_TToken[nNotRunThread].m_hThread,
                    pThis->m_TToken[nNotRunThread].m_nThreadID);
                //如果没有启动起来,下轮再说，此处不再报错
            }
        }
        Sleep(OPEN_THREAD_DELAY);    //一定要等够 250ms
    }
    XMD(pThis->m_nThreadPoolThreadCount);    //退出时，线程计数器-1
#ifdef WIN32                                //按照 Linux 和 Windows 两种方式退出
    return THREAD_POOL_EXIT_CODE-1;
#else // not WIN32
    return null;
#endif
}

```

9.2.4.3 守候线程释义

在前文，笔者曾提出过“守候线程”的概念，但是没有细讲，可能部分读者比较费解。其实，线程池内的“管理者线程”和“服务者线程”都是典型的“守候者线程”。

所谓“守候线程”，就是线程启动后，以一个死循环形式，不断检索一个或者几个变量的值，并根据这个值，执行具体的功能，如果该值表示目前无任务可做，则守候线程处于一种长时间睡眠状态，中间做间歇性苏醒检查。

这在商用并行工程，是很常见的一种开发方式。具体实现，可以用线程完成，也可以用进程完成。比如 Windows 下检测 USB 设备的插入或拔出，做动态通告，就是一个系统级的守候进程在完成。Linux 下也有虚拟内存维护等守候进程在工作。

各位读者以后在开发商用并行工程的过程中，可能会频繁遇到类似的需求，很多时候，程序员脑子里可能由于没有这个守候的概念，不知道可以单做一根线程，就检索某一个状态，并以此触发动作，造成开发不便。

这里笔者给出一个提示，只要资源允许，一般说来，针对某一个需要长期观测的状态值的检测，可以使用一根独立的线程来完成，只要线程中做了足够的睡眠，并选择了合适的睡眠周期，这种设计并不会给系统造成太多不必要的负担。

守候线程的设计实例，很象前文所述的基本任务，即“初始化代码+循环体+结束代码”的结构。

9.2.4.4 “快照”的原理

在管理者线程的实例中，大家可能对一句话比较费解，这里特别说明一下。

```
nIdleThread=XMG(pThis->m_nThreadPoolIdleThreadCount);
```

大家可能注意到，类成员变量 `m_nThreadPoolIdleThreadCount` 本身存储的就是当前处于 Idle 状态的线程统计数，应该可以直接拿来使用，为什么一定要用一个中间变量 `nIdleThread` 来做一个求值动作呢？

这是因为在多任务环境下运行的程序，由于每时每刻都有多个执行生命体在运行，因此，一些关键的共享变量，会发生非常频繁地更动，比如这个 Idle 线程统计器，就是这类变量。

虽然我们能使用多线程安全的变量类型，保证了其每一次读出，或者修改的动作，不会被打断，访问本身是安全的，但是，我们还要关注一个细节，就是我们如果在一个程序段中，需要多次参考这个变量的值，它的值在这次运算过程中，会不会改变，并由此引发逻辑歧义，直到产生 bug。如下面这段函数举例：

```
CMint g_nCount;           //全局安全变量，可能被其他线程改变
void Func(void)
{
    if(g_nCount.Get())      //---1
        //...
    //...
    int i=g_nCount.Get();   //---2
}
```

大家可以看到，在 `Func` 中，一段计算两次访问了全局安全变量 `g_nCount`，但是，虽然 `g_nCount` 是线程安全的变量，可以保证访问安全，但是，从行数 1 到行数 2，中间并没有加锁，因此，可能会被打断，此时，`g_nCount` 极有可能被其他线程改变。

换言之，对于同一个计算，`g_nCount` 的两次读取，很可能不是同一个值。这在高速并行计算流程中很常见。

通常，这种改变并不会导致什么 bug，但是，以防万一。在笔者过去多年的开发经历中，确实有那么一次两次，由于这个变量在同一笔计算中，有两个值，导致了 bug，并且，还是很不好查找的 bug。

因此，建议各位读者以后遇到类似问题，一律使用“快照”原理来完成程序设计。如上例可以改写成如下模型：

```

CMint g_nCount;
void Func(void)
{
    int nCount=g_nCount.Get();    //----1
    if(nCount)
        //...
    //...
    int i=nCount;
}

```

注意标为 1 的这一行，我们在此定义了一个中间局部变量 **nCount**，以此一次性获取 **g_nCount** 的当前值，这相当于为该值拍了一个快照，以后，本次计算中，统一使用 **nCount** 参与计算，不再重新获得 **g_nCount** 的值，这保证了一个计算元，在一次计算中的一致性，避免了潜在的 **bug** 危险。这就是“快照”原理。

在多任务的并行计算中，在商用数据传输工程的开发实战中，这类“快照”手法是很常见的，主要的目的就是上面所述的原因。

本小节介绍的服务者线程，虽然 **nIdleThread** 只用了一次，但是笔者习惯性地仍然使用了“快照”手法，避免将来某个时刻，笔者修改代码，需要多次使用这个值时，引发潜在的 **bug**。

提示：扩展一点说，商用并行工程中，由于到处是多任务并行运行，很多时候，各个关键值都是不断变化的，因此，应该大规模采用“快照”思维，即每当需要使用一个全局的，多任务共享的变量时，都要有意识地使用一个本地局部变量将其拍一个快照来使用，而不能直接访问，否则容易引发逻辑歧义等潜在的 **bug**，导致不必要的麻烦。

9.2.5 服务者线程

服务者线程是线程池的另外一个核心，可以说，线程池所有的具体功能实现----为各种注册任务提供时间片服务，都是通过服务者线程来完成的。

服务者线程的逻辑如下：

1、启动后，检测任务区，如果没有任务，则是 **Idle** 状态，睡眠一段时候，重复检测，这也是一个守候线程的行为，即不断检测任务区块的数据变化，并根据这种变化决定行为，发起动作。

2、如果检测到一个任务，则执行之，执行完毕后，清空任务，保证每个任务有且仅有一次执行机会。

3、**Idle** 状态，还要检测一下目前属于 **Idle** 的兄弟线程的总数，如果超标（超过 10 条），则退出自己。一次保证 **Idle** 数在 10 条左右浮动。可以说，**Idle** 线程数从 0~10 条的增量变化，是由管理者线程负责，但是，从 11~10 条的减量变化，是由服务者线程自行完成的。

通常，这两个动作都总是在互动，因此，目前系统的线程数看起来，会经常在 10 和 11 条之间跳变。

```
//服务线程
THREADFUNCDECL(CTonyThreadPool::ThreadPoolThread,pParam)
{
    //由于历史原因，此处有一点歧义，此处的 pThis 不是线程池对象指针
    //而是任务块的指针，由于线程池开发较早，出于“尊重”原则，此处没有做更动。
    //线程池对象指针，此处表示为：pThis->m_pThreadPoolObject
    SThreadToken* pThis=(SThreadToken*)pParam;
    //刚启动，设置为 Idle 状态
    XMS(pThis->m_nState,TPOOL_THREAD_STATE_IDLE);
    //线程计数器+1
    XMA(pThis->m_pThreadPoolObject->m_nThreadPoolThreadCount);
    //Idle 线程计数器+1
    XMA(pThis->m_pThreadPoolObject->m_nThreadPoolIdleThreadCount);
    //注意这个守候循环，检索了 bThreadContinue 参量，以便配合最终的退出动作
    while(XMG(pThis->m_pThreadPoolObject->m_bThreadContinue))
    {
```

```

//取任务
switch(XMG(pThis->m_nState))
{
case TPOOL_THREAD_STATE_NOT_RUN:
    //这个状态表示没有线程为任务块服务，但现在本线程已经启动
    //如果仍然看到这个状态，肯定出错了，表示外部程序状态设置不对
    //但这个错误不严重，自动进那个修补一下就 OK
    //修补的方法就是设置为 Idle 状态
    XMS(pThis->m_nState, TPOOL_THREAD_STATE_IDLE);
    //注意此处没有 break
case TPOOL_THREAD_STATE_IDLE:          //没有命令
default:
    //这是正常睡眠
    break;
case TPOOL_THREAD_STATE_BUSY:          //Register 下任务了
    //请注意一个细节，这里没有把 Idle 计数器-1，
    //这是因为 Register 函数做了这个动作，后文有详细描述
    if(pThis->m_pCallback) //检查是不是真的有任务
    {
        //将执行权交给新的任务（一次），请注意参数传递
        pThis->m_pCallback(pThis->m_pCallParam,
            pThis->m_pThreadPoolObject->m_bThreadContinue);
        //空闲线程数+1
        XMA(pThis->m_pThreadPoolObject->m_nThreadPoolIdleThreadCount); //----a
    }
    break;
};
//检查空闲线程总数
if(WHILE_THREAD_COUNT< //----b
    XMG(pThis->m_pThreadPoolObject->m_nThreadPoolIdleThreadCount))
    break; //如果备用线程超出限额，则跳出死循环，退出自己
//所有工作做完，把自己置为 Idle 状态
if(TPOOL_THREAD_STATE_IDLE!=XMG(pThis->m_nState)) //----c
    XMS(pThis->m_nState, TPOOL_THREAD_STATE_IDLE);
Sleep(DEFAULT_THREAD_SLEEP); //睡眠，等待下次任务
}
//退出流程
//Idle 计数器-1
XMD(pThis->m_pThreadPoolObject->m_nThreadPoolIdleThreadCount); //----d
//线程总计数器-1
XMD(pThis->m_pThreadPoolObject->m_nThreadPoolThreadCount);
//把任务区块的状态设置为正确的值（没有线程为其服务）
XMS(pThis->m_nState, TPOOL_THREAD_STATE_NOT_RUN);
#ifdef WIN32
    return pThis->m_nExitCode; //两种方式退出
#else // not WIN32
    return null;
#endif
}

```

这个函数最容易出错的细节在于标注为 a、b、c 这三行的关系。笔者在书写这段代码时，就发生过错误。因为这里面有个很“绕”的环节必须关注。

大家注意到，在检测到有任务时，本函数并没有把 Idle 计数器-1。这是因为后文的注册函数已经在注册时，做过-1 运算了，因此，执行完毕后，必须立即在标注为 a 这一行，做一次+1 计算。

这是因为，后文标注为 b 的检测逻辑，是 switch 语句中，TPOOL_THREAD_STATE_IDLE 和 TPOOL_THREAD_STATE_BUSY 两种状态通用的，因此，它的默认条件是，有多少 Idle 线程，Idle 计数器就应该是多少，否则检测不正确。

而我们的服务线程，刚刚从 Busy 状态返回时，m_nThreadPoolIdleThreadCount 的计数器一定是比真实的 Idle 线程少 1 的，如果不做 a 行的加法计算，则 b 行的检测，面临两种先决条件，从 TPOOL_THREAD_STATE_IDLE 这个分支下来，m_nThreadPoolIdleThreadCount 和真实的 Idle 线程数相同，因为没有变化，而从 TPOOL_THREAD_STATE_BUSY 这个分支下来，m_nThreadPoolIdleThreadCount 比真实 Idle 的线程少 1，这就造成判断失误。

提示：这是程序开发中较容易被忽视的一个细节，当我们从一个大型判断语句，或者一个大型 switch 语句中跳出，应该考虑到，每个分支走下来，对每个相关变量的影响都是相同的，否则会后续代码引发 bug。这种情况需要程序员仔细甄别，小心设计，预防 bug。

9.2.6 注册函数

注册函数是线程池提供公共服务的关键，所有的任务注入，由此进入。注册函数主要的功能就是遍历所有的任务区块，选择其中出于 Idle 状态的任务块（这表示其有线程服务，且目前没有任务），然后将任务注入，等待运行。

值得关注的是，这种注入可能会失败，假如某个时刻，找不到 Idle 状态的任务块，就可能失败。这在应用程序刚开始运行时，很常见，管理者线程按照 250ms 的步距启动线程，很可能还来不及启动这么多服务者线程，就可能找不到 Idle。

但这种状态会随着时间的改变而好转，因为管理者线程总是不断检测任务块，如果 Idle 的任务块少于 10，就会不断启动新线程，大约过几百个毫秒，就可能等到 Idle 线程的出现。

因此，注册函数有多个构型，对外也提供一些和缓的等候手段，可以稍微将应用程序的注册行为“悬挂”一会，等有足够的 Idle 线程时，注册成功再返回。

由于注册函数的状态不确定，因此这里定义了几个显式的状态返回码，帮助应用程序识别注册结果。

```
#define _THREADPOOL_CAN_NOT_USE    -2 //线程池未初始化，无法工作
#define _THREADPOOL_OVERFLOW       -1 //线程池溢出标志，无法注册
#define _THREADPOOL_PLEASE_WAIT    0  //线程池目前没有备用线程，请等待
#define _THREADPOOL_OK              1  //线程池注册成功
```

9.2.6.1 GetIdleThread

GetIdleThread 是最核心的一个功能函数，其逻辑非常简单，就是在线程池任务数据块中，寻找出于 TPOOL_THREAD_STATE_IDLE 的数据块，返回指针备用。

```

//获得一个空闲的线程编号
//如果无空闲,则返回-1
int CTonyThreadPool::GetAIdleThread(void)
{
    int nRet=-1;
    int i;
    for(i=0;i<THIS_POOLTHREAD_MAX;i++)    //遍历检索
    {    //注意, 仅仅检索 Idle 一个状态
        if(TPOOL_THREAD_STATE_IDLE==XMG(m_TToken[i].m_nState))
        {
            nRet=i;
            break;    //找到跳出
        }
    }
    return nRet;    //没找到, 可能返回-1
}

```

9.2.6.2 ThreadPoolRegisterANewThread

本函数是真实的注册行为, 后面的复杂逻辑注册函数, 均是调用本函数完成。本函数是一个“不保证服务”的函数, 主要原因就是注册成功与否, 取决与目前是否有 Idle 的服务线程存在, 因此, 不一定会成功。通过返回码, 通知调用者注册情况。

```

//注册一个新任务
//请注意,这里有临界区保护,线程安全
//注册成功,返回_THREADPOOL_OK, 否则返回其他错误码
int CTonyThreadPool::ThreadPoolRegisterANewThread(
    _TPOOL_CALLBACK pCallback,           //任务回调函数指针
    void* pParam)                        //透传的任务参数指针
{
    int nRet=_THREADPOOL_PLEASE_WAIT;    //返回值设置初值
    MUTEXLOCK(&m_RegisterLock);          //加锁
    int nIdleThread=GetAIdleThread();    //取得 Idle 的线程编号
    if(0>nIdleThread)
    { //没有找到 Idle 服务线程
        if(THIS_POOLTHREAD_MAX==XMG(m_nThreadPoolThreadCount))
        {
            //没有空闲线程,而线程总数已经达到上限,返回 OverFlow 标志
            nRet=_THREADPOOL_OVERFLOW;
        }
        else
        {
            //没有空闲线程,仅仅是还没有来得及开启,请调用者等待
            nRet=_THREADPOOL_PLEASE_WAIT;
        }
    }
    else
    {
        //找到空闲线程,添加任务
        m_TToken[nIdleThread].m_pCallback=pCallback; //这里可以看出如何将
        m_TToken[nIdleThread].m_pCallParam=pParam;  //任务添加到任务区块
        XMS(m_TToken[nIdleThread].m_nState,         //注意,本任务块被设置为
            TPOOL_THREAD_STATE_BUSY);               //busy,因此,不会被再次
                                                    //添加新的任务
        XMD(m_nThreadPoolIdleThreadCount);          //前文所述,Idle 计数器-1
        nRet=_THREADPOOL_OK;                        //返回成功标志
    }
    MUTEXUNLOCK(&m_RegisterLock);              //解锁
    return nRet;
}

```

9.2.6.3 ThreadPoolRegisterANewThreadWhile

通常为了简化应用层操作,应用层需要一个确定的注册成功结果,哪怕为此等待一会也可以,因此,在上一个不保证服务的注册函数基础上,笔者增加了这个带延时等待,尽量注册成功的逻辑函数。


```

//一定要注册成功,可以等待新的 Idle 线程被管理者线程启动,除非遇到 OverFlow,否则循环等待
int CTonyThreadPool::ThreadPoolRegisterANewThreadWhile(
    _TPOOL_CALLBACK pCallback,           //任务回调函数指针
    void* pParam)                       //透传的任务参数指针
{
    int nRet;
    while(1)        //死循环等待
    {    //调用上一函数,开始注册
        nRet=ThreadPoolRegisterANewThread(pCallback,pParam);
        //注册成功,或者线程池溢出,都返回
        if(_THREADPOOL_PLEASE_WAIT!=nRet) break;
        Sleep(OPEN_THREAD_DELAY);        //最多等一会,新的线程就已经建立了
    }
    return nRet;
}

```

9.2.6.4 ThreadPoolRegTask

这是公有方法函数,目的是调用前述函数,为应用层提供注册功能调用。其中 bWait4Success 默认值为真,就是一定要等待成功,应用程序也可以根据需要,设置为假,不等待,直接返回。

```

int CTonyThreadPool::ThreadPoolRegTask(
    _TPOOL_CALLBACK pCallback,           //任务回调函数指针
    void* pParam,                       //透传的任务参数指针
    bool bWait4Success)                 //是否需要等待注册成功
{
    if(bWait4Success)                   //根据标志,调用不同的函数
        return ThreadPoolRegisterANewThreadWhile(pCallback,pParam);
    else
        return ThreadPoolRegisterANewThread(pCallback,pParam);
}

```

9.2.7 线程池小结

线程池是笔者早期的代码模块,其中有很多理念尚不完全符合 C 和 C++ 无错化代码设计方法。但是,无可否认的是,经过这么多年的工程实践,线程池这个模块已经被验证为一个很稳定的模块。

在本章后几节的任务池等模块中,都是在线程池的基础上做二次开发完成的,线程池作为笔者商用工程库的线程管控核心,一直稳定地运行于各个工程中,起到重要的作用。

线程池的整理程序书写,其实比较简单,但是,请各位读者关注,线程池体现出来的多任务开发原则,“主动池”开发特点,都是比较新颖的思路,请各位读者务必仔细体会,后文中,在进行传输实战的介绍中,我们会大量使用线程池体现出来的思路。

最后,笔者还是要给出一个线程池的通用调用模型,请大家参考。

9.2.7.1 线程池调用示例类声明

这里,笔者使用一个非常接近工程实战的代码,来向大家展示如何使用线程池。

```

//线程池测试调用示例类
class CTestThreadPool
{
public:
    CTestThreadPool();
    ~CTestThreadPool();
private:
    //线程池的回调函数，也就是应用程序的线程函数
    static void TestThread (void* pCallParam,MBOOL& bThreadContinue);
private:
    CTonyLowDebug* m_pDebug;                //Debug 对象指针
    CTonyThreadPool* m_pThreadPool;          //线程池对象指针
    //这是比较特殊的一个例子，虽然线程池提供了最终的线程安全退出手段
    //但是，我们要知道，很多时候，我们的一个应用类，和线程池的生命周期并不一致
    //通常情况下，应用类生命周期较短，因此，应用类应该有自己的线程安全退出手段
    //因此，这里仿造线程池的设计，也声明了两个线程安全的管理变量
    CMbool m_bThreadContinue;
    CMint m_nThreadCount;
};

```

9.2.7.2 线程池调用示例类构造函数

在构造函数中，我们需要创建 debug 对象，线程池对象，并且注册线程。

```

CTestThreadPool::CTestThreadPool()
{
    m_bThreadContinue.Set(true);                //线程持续标志置为真
    m_nThreadCount.Set(0);                      //线程计数器置为 0，这些是赋初值动作
    m_pDebug=new CTonyLowDebug("", "", true);    //创建 debug 对象
    m_pThreadPool=new CTonyThreadPool(m_pDebug); //创建线程池对象
    if(m_pThreadPool)
    {
        //如果一切 OK，则开始注册任务线程回调。
        m_nThreadCount.Add();                    //注意，线程启动前，计数器先+1
        //注册线程函数，注意，由于线程函数需要访问本类的管理变量，这里把 this 指针传入
        m_pThreadPool->ThreadPoolRegTask(TestThread,this);
    }
}

```

9.2.7.3 线程池调用示例类线程函数

线程函数基本上是一个示例空函数，大家可以看看基本的管理要素。

```

//测试线程，注意其完全符合线程池回调函数构型，且是静态函数
void CTestThreadPool::TestThread(void* pCallParam,MBOOL& bThreadContinue)
{
    //强制指针类型转换，获得本对象指针
    CTestThreadPool* pThis=(CTestThreadPool*)pCallParam;
    while(XMG(bThreadContinue))                //这是标准写法，即线程池退出支持
    {
        //注意这一句，一个循环可以有多个退出点
        //此处增加这一句，是支持本对象的退出请求
        if(!pThis->m_bThreadContinue.Get()) break;
        //...程序逻辑
        Sleep(MIN_SLEEP);                      //习惯性睡眠
    }
    pThis->m_nThreadCount.Dec();                 //注意，退出时计数器-1
}

```

9.2.7.4 线程池调用示例类析构函数

析构函数，首先会使用线程安全退出方式，退出本对象的线程，然后摧毁各个成员对象，如线程池和 debug 对象。

```
CThreadPool::~CThreadPool()
{
    //请注意，这里是利用本类的线程控制变量，退出本类的线程
    //从本例的逻辑看，这个设计稍微有点多余，因为线程池随后的摧毁动作，也会退出线程
    //但请注意，在世纪工程中，线程池往往是全局唯一，是外部传入，因此
    //原则上，每个开启线程的应用类，都应该有自己的线程控制手段
    //在本对象关闭或摧毁时，必须安全退出自己所属的所有线程，方能防止 bug。
    m_bThreadContinue.Set(false);          //标准退出逻辑
    while(m_nThreadCount.Get()) {Sleep(MIN_SLEEP);}
    if(m_pThreadPool)                       //摧毁线程池
    {
        delete m_pThreadPool;
        m_pThreadPool=null;
    }
    if(m_pDebug)                           //摧毁 debug 对象
    {
        delete m_pDebug;
        m_pDebug=null;
    }
}
```

9.2.7.4 线程池调用示例类说明

上述线程池的调用实例，实际上是笔者工程中常用的手法，每个类，依赖线程池对象，注册启动自己的线程，在线程池已经保证了所有线程能安全退出的前提下，原则上，每个类自己也必须维护一套线程控制变量 `bThreadContinue` 和 `nThreadCount`。

这样，当本对象关闭，或者析构时，可以利用这两个变量，将本对象所属线程全部退完，确保安全运行。

```
m_bThreadContinue.Set(false);          //标准退出逻辑
while(m_nThreadCount.Get()) {Sleep(MIN_SLEEP);}
```

笔者再补充一点，事实上，在工程中，很多核心关键线程，其 `while` 主循环内部往往有很多 `if` 语句的判断，都是退出条件，工程实战中，一个线程通常都是多条件退出，甚至有时候，还有个全局的 `g_bMainExitFlag`，在通知所有的线程退出，这些都是有可能的。这没有什么道理，仅仅是经验，供大家参考。

9.4 任务池

如果说，线程池使我们第一次可以精确、安全地实现线程控制，彻底解决线程起停带来的一系列潜在的 `bug` 风险，那么，任务池就是在线程池基础上，一次大规模的升华和改进。是笔者在深入理解多任务开发环境和并程序开发之后，对所有并行任务做出进一步抽象的产品。

任务池的出现，是经过了长期的开发思考，目的是一劳永逸地解决多任务并行环境下线程总数的限制，以及商用数据传输工程中，高吞吐量，高并发的需求。

任务池的开发，大规模使用了前文所述的锁、队列和线程池的概念，并在这些模块的基础上开发完成。

9.4.1 任务池的原理分析

在笔者设计出线程池模块后，常见的多线程开发任务已经变得很轻松，但是，不久之后，笔者发现，线程池还有很多不足：

1、线程池只能使用固定线程数，而这个线程数受到操作系统的限制，比如在 Linux 下，一般一个进程内部，线程总数不建议超过 300 条。这就限制了一个应用可能引入的总的并发任务数，不能超过 300 个。

2、即使没有操作系统的限制，在一个 CPU 的操作系统下，一个进程能并发的总的线程数也是有限的，太多的线程，会导致操作系统的任务链表过长，遍历效率过低，严重影响系统总的计算效率。

3、但在商用数据传输工程实践中，我们的一个大规模服务器模型，通常需要处理成千上万个并发的客户请求，上述限制，显然远远不够。

4、另一方面，线程池的任务体，即任务回调函数的编写，基本上是典型的“守候线程”编写方法，主要逻辑就是“检测---处理---睡眠”，而通常运行中，睡眠占据了大部分线程时间。这表示我们从操作系统申请的时间片，大多数又还给了操作系统，对我们应用程序而言，这是一种“浪费”。

因此，为了解决上述矛盾，大幅度提升我们的应用程序并发的任务数，使多任务开发，在商用数据传输工程中，真正能起到支撑作用，笔者考虑，在线程池基础上，需要进一步提升和升华，实现更深入的设计。

9.4.1.1 线程池的抽象运行模型

在线程池中，我们知道，有多少个任务数据块，理论上，就允许多少个操作系统的线程（执行生命体）并发存在。如下图所示：



图 9.11：线程池的抽象逻辑示意

这里我们可以看出，线程池任务数据块和线程执行生命体一一对应，与应用程序注册的任务，也是一一对应，因此，总的并发任务数，受到线程池极限的限制。无法满足大并发量的需求。

9.4.2.2 问题出在哪里？

前文我们讨论了一个基本的任务函数，应该是什么样子，基本上由三部分组成，“初始化代码---循环体---结束代码”，大家可能注意到，这和我们大多数时候，开发单线程的任



图 9.12：单线程任务模型

务函数，并没有什么不同。如下图：

这里我们可以看出一点问题，虽然我们的线程池已经深入利用了多任务并行处理能力，我们为多个任务申请了时间片，并实现的动态的管理，但是，针对线程池每个任务而言，我们为其申请的一组时间片，如果密集排列起来看，其实还是一个单任务运行模型。

在前文，线程的抽象理解一节中，我们有一个结论，也说明了这个问题。线程运行的核心特点，就是物理上的片段性，以及逻辑上的连续性。如下图：

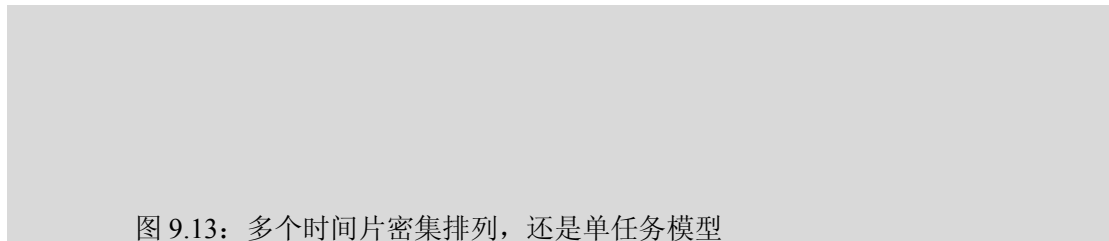


图 9.13：多个时间片密集排列，还是单任务模型

大家注意到没有，线程池内的服务线程，从其生命周期来说，还是单任务模型，其处理的每个任务是没有交集的，换言之，一旦一个任务启动，服务线程就将执行权交给了该任务的回调函数，在回调函数返回前，服务线程是没有能力处理第二个任务的。

而前文我们说过，由于应用程序的任务函数，通常内部都是一个带 Sleep 的死循环，此时，大量的时间片被浪费在无意义的睡眠上，如果我们把线程池的一条服务线程，看做一台计算机的话，又出现了单任务 DOS 模型下的情况，大量的计算资源被浪费，这也是线程池服务效率不高的主要原因。

我们再来看看线程池的实例，就会更加清晰这一点：

```

void CTestThreadPool::TestThread(void* pCallParam,MBOOL& bThreadContinue)
{
    CTestThreadPool* pThis=(CTestThreadPool*)pCallParam;
    while(XMG(bThreadContinue))
        if(!pThis->m_bThreadContinue.Get()) break;
        //...程序逻辑
        //问题就出在这一句，我们的线程，接管了时间片序列，获得了执行生命体
        //但是，当我们无事可干的时候，我们在睡眠，也就是把时间片又归还给操作系统了
        //但是，这种 Sleep 又必不可少，否则，CPU 的占用率会居高不下。
        Sleep(MIN_SLEEP);
    }
    pThis->m_nThreadCount.Dec();
}

```

那么，我们是不是得出一个结论：**只要是按照这类调用者交付执行权的执行模型，就是低效率的，存在计算资源浪费的模型。**

9.4.2.3 多任务的新世界观----“我管”还是“你管”

这个问题困扰了笔者很久，直到有一天，笔者回顾当年在任天堂游戏机上的开发实例，才发现一个没有想到的关键点。

前文我们说过，任天堂游戏机的多任务机制，是依靠中断来实现的。这就说明了一个问题，只要开机运行，中断就是连绵不绝的，不断产生的。

我们前面的讨论中，讨论过一点程序的基本构型，基本上，一个要完成实际意义功能的函数、模块、应用程序，其实都是以大大小小的循环嵌套而成，大家抽象一点理解，看是不是这样？

笔者发现自己的一个误区，长期以来的单任务开发模型，其实还是深深地影响笔者的开发思路。最关键的，就在这个循环嵌套上。

包括笔者在内，几乎所有的程序员，在开发时会养成一个惯性思维，就是“**一切自己来**”。我们在实现一个模块，一个应用的时候，我们习惯考虑所有的事物，我们习惯于自己设计变量，并为其赋初值，我们习惯自己构建大大小小的循环逻辑，直到完成所有的工作。这说明了，我们长期以来，习惯于“**我管理一切**”。

但是，多任务模型不一样，多任务模型下，循环体天生就存在，任天堂游戏机的中断，本次结束后，零点几个毫秒之后，就又会发起，这和有没有我们的程序没有关系，即使我们给游戏机不灌注任何程序，它只要开机，IRQ、NMI 中断就永无止境的发生。这好比一个饭店，不管有没有客人住，但是，饭店的房间始终是存在在哪里的。

多任务操作系统的运行环境其实也差不多，不管我们是否启动进程、线程，不管我们是否注册任务，多任务操作系统一启动，就有一个大的中断循环，以及一个任务链表在工作，这是永无止境的工作，直到关机。即使我们不启动任何应用程序，操作系统的这一特点是不会改变的。

其实，这里我们可以看出计算机的一个特点，计算机内部，总是有大大小小的循环在运行，当没有任务时，就是空循环，有任务时，就是任务循环，多任务操作系统如此，单任务的 DOS 也是如此，大多数时候，DOS 操作系统内部的主循环，在等待键盘的输入。

ok，既然我们大多数程序逻辑，都是一个循环，而操作系统本身就提供了循环模型，不管我们承不承认，也不管我们用不用，操作系统总是有个循环在哪里工作，这样，我们是不是可以得出一个结论，“**为什么我们还要自己管理循环？**”

从程序设计角度，只要我们使用 switch 语句，利用一定的技巧，完全可以不再自己管理循环，而大大方方交由底层的操作系统，甚至我们的线程池来做循环，我们只要知道，我们的程序，每隔一段时间，一定会被重入执行，就够了。

我们来看下面这个程序示例，首先，我们先写一个基础的函数，然后试图改写成使用操作系统循环的例子。

//这是一个通常意义上的程序逻辑，围绕一个主循环工作。

```
void AppFuncOld(void)
{
    //...  init 代码
    while(1)
    {
        if(...) break;           //当某种条件符合时，跳出循环，结束逻辑
        //...
        Sleep(MIN_SLEEP);
    }
    //...  exit 代码
}
```

这是一段很常见的程序逻辑，大家只要写过几年程序，应该很熟悉这类逻辑。基本上就是前文所述的“初始化代码+主循环体+结束代码”结构。

其中特别要关注的是，初始化代码和结束代码只执行一次，而循环体内的代码，根据程序逻辑，可能执行 1~n 次。这种 1+n+1 的执行开销，也就是我们常见的程序开销。

现在，我们用一个循环函数来模拟操作系统的循环，叫做 OSLoopDemo。

```
void OSLoopDemo(void)
{
    while(1)           //永远不断的死循环
    {
        AppFunc();     //假定只有一个任务，就是 AppFunc
        Sleep(MIN_SLEEP); //就算是操作系统的循环，也要遵循睡眠原则
    }
}
```

这里面体现出两点：首先，操作系统提供执行机会，即每隔一段时间，就会去执行一次应用程序代码 AppFunc，至于里面是什么内容，操作系统不管。其次，操作系统的这个主循环，就是一个包含了 Sleep 的死循环，永远不断，直到关机。

现在，我们看看 AppFunc 要怎么书写，才能实现改写前的逻辑。

```

void AppFunc(void)
{
    //nStatus 是最关键的，函数状态机变量，记录了函数目前的运行状态
    //注意，这是一个静态变量，函数退出，变量值还存在，不会被摧毁
    //当然，也可以使用类成员变量，或者全局变量等生命周期长于函数本身的变量代替。
    static nStatus=0;
    switch(nStatus)          //这就是执行机，根据状态机，决定运行的程序段落
    {
        case 0:
            //...  init 代码
            nStatus++;        //注意，状态机+1，表示下次进入，开始执行下一段代码
                               //换言之，程序的顺序向下执行，是通过状态机的累加
                               //实现的。

            break;
        case 1:              //循环体代码，注意，while 已经不见了
            {
                //...
                if(...)       //这是跳出条件，上文中的 break 已经被状态机+1 代替
                    nStatus++;

            }
            break;
        case 2:
            //...  exit 代码
            nStatus++;        //注意，exit 代码执行完毕后，状态机立即+1，表示
                               //exit 代码也只执行 1 次

            break;
        default:
            break;            //以后没有代码执行，实际上也就是函数逻辑完成
                               //当然，实际操作系统中，到此，可能就把本回调
                               //从链表中摘除，不再执行本逻辑了。
    }
}

```

这就是典型的“你管理”模型，这个“你”，可能是操作系统，可能是我们的线程池中的线程，也可能就是本例的 OSLoopDemo。请注意本例的几个要点：

- 1、程序不再是传统意义上的顺序执行，被程序员根据业务逻辑，人为打断成几个片段（switch 中的几段）。
- 2、如果程序要向下执行，必须把状态机 nStatus+1，那么，下次进入的时候，执行机根据新的状态机值，会跳到下一段执行。
- 3、如果状态机不改变，就是死循环，每次进入，都是执行同一段逻辑，如本例中的循环体。
- 4、由于 C 和 C++ 无错化程序设计严禁 goto 向前跳转，因此，状态机-1 原则上是不允许的。
- 5、本例证明：我们利用一个外部循环，是完全可以实现所有的程序逻辑的。

这应该说，是笔者程序设计思维的一个重大突破，在想通了这个道理之后，笔者开发中，再也不局限于自己管理一切，而是有意识地利用系统资源，把一些关键的控制逻辑交给系统来控制，以此将选择权最大限度地返回给操作系统，使操作系统的多任务调度，更加和谐，合理。

如果说，C 和 C++无错化程序设计方法，是从程序书写的角度，保证了程序的安全性，降低了 bug 率的话，那么，这个“你管”思维，则是从操作系统宏观运行环境的角度，保证了应用程序设计的合理性，运行的温和性，保证了程序的运行安全性，同样，也大幅度降低了 bug 率。

9.4.2.4 任务池的理想模型

当我们做了从“我管”到“你管”这么大的一个思路转变之后，我们回头再看线程池，发现确实问题就比较多了。

线程池的核心设计逻辑，还是属于“守候线程”的思路，前文我们讲过，守候线程，就是安排一个线程，不断循环检测，查看某一个或某几个状态的值是否发生改变，并根据这些改变，引发一些新的动作，形成一种“动作源”的效果。Windows 程序的消息循环就是这种开发思路的典型例子，有事件，引发动作，无事件，死循环。

应该说，这种逻辑，是多任务并行开发的一个普遍现象，大多数时候，我们几个线程协同完成一个业务计算的概率还是很小，通常都是每个线程守候一个事件，各自负责一笔独立业务。比如，应对多个并发客户请求的网络服务器，通常一个线程负责一个客户的所有事务，不同线程间，业务上的交集很少，这也是降低程序逻辑复杂度，降低开发难度的通常手法。

但是，线程池建议的“守候线程”模型，体现了一种典型的“我管理一切”的设计思路，这导致了线程池先天性的效率低下，必须予以改变。

在改变之前，我们必须先思考一个问题，多任务操作系统是怎么做的？

在前文我们讨论过，多任务操作系统其实就是利用硬件系统的中断，获得一个不间断的中断循环执行生命体，然后通过一个链表管理，来分配时间片，最终实现多任务并发的执行效果。

那这个问题就简单了，我们为什么不能仿造操作系统的工作，在我们线程池的基础上，实现二次的时间片分配？

我们知道，当我们应用程序，在线程池的支持下，注册启动了一根线程，那么，我们相当于获得了一组时间片的执行权，这一组时间片可能中间会被其他执行生命体打断，但从逻辑上看，我们还是连续执行的，也就是说，**线程池内的一根线程，就像一台单任务操**

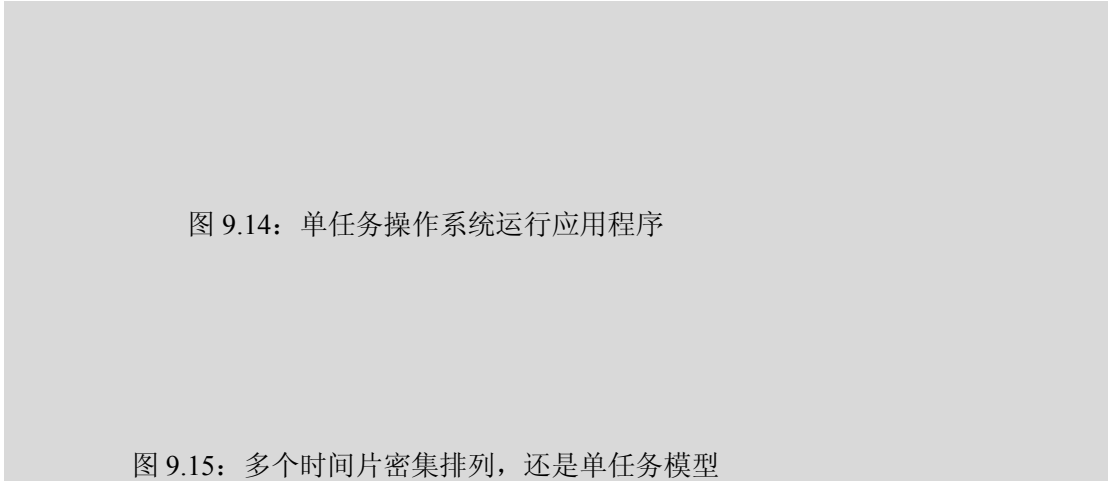


图 9.14：单任务操作系统运行应用程序

图 9.15：多个时间片密集排列，还是单任务模型
作系统的计算机。我们来看看下面两个图，有什么差别？

既然我们已经讨论了，在单任务的计算机上，我们能实现多任务操作系统，我们当然也能按照这套机制，实现我们自己的时间片二次分配逻辑，也就是在应用程序内部，我们构建了第二层多任务执行内核。从而，使我们的应用程序，天生就是多任务模型开发的。

因此，我们的改变的目标，就是希望重新整理线程池内申请出来的线程的执行逻辑，

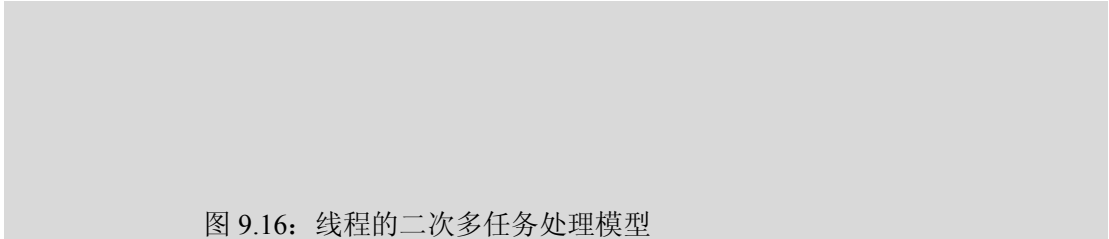


图 9.16: 线程的二次多任务处理模型

将其固有的单任务执行模型，改变成多任务执行模型，如下图所示：

那么，我们究竟怎样做到这一点呢？其实这个答案操作系统已经告诉了我们了，我们如果以一个链表形式，为一个线程构建一个任务队列，线程不断依次轮询，有任务就不断执



图 9.17: 线程任务链表模型

行即可。如下图：

大家可以看到，这个模型和操作系统何其相似。其实，从单任务模型转化到多任务模型，添加这个链表，也就足够了。

不过，这里我们还要多讨论一点问题。

我们知道，通常一个应用程序的线程池，有多个线程在并行运行，如果我们为每个线程都构建这个队列，一来没有必要，二来占用资源，三来还可能带出动态负载均衡等一系列问题。那么，我们可不可以让一个任务池，就只有这么一个链表，所有的线程都来这个链表中取任务，谁有空谁来做事，行不行？答案是当然可以，笔者的任务池就是这么实现的。

笔者经过实践发现，这也是任务池的一种比较理想的模型，可以最大灵活度地分配任务。如下图：



图 9.18: 任务池动态调用模型

如图所示，任务池有一个任务链表，同时从线程池申请多个执行生命体---线程，每个线程的逻辑非常简单，不断检索任务链表，如果有，就执行一次，并且归还到任务链表末尾，（除非逻辑决定该任务已经执行完毕，无需再执行，则从链表删除）。

这样，线程数可以根据需要，实际确定，任务链表的长度，可以根据需要，实际确定，每一个逻辑都是异步的，在保证灵活性的前提下，又保证了高效性，实现了较完美的任务时间片分配逻辑。

9.4.2 任务池的需求和设计

当我们思考出任务池的原理后，我们可以考虑如何实现一个真实的任务池。不过，在实战之前，我们还有一些细节需要讨论。

9.4.2.1 任务的粒度

在上一小节线程池的介绍中，我们引入了“任务”这个概念，那么，任务池的“任务”，和线程池的“任务”，有什么不同呢？

我们首先澄清一个细节，虽然任务池致力于取消线程的“大守候循环”，但这并不是说，以后任务池中的任务，不允许有任何循环，那就太形而上学了。

我们看过上文的例子，虽然 `AppFunc` 的主要循环逻辑，可以通过一定的改写，巧妙地利用操作系统的时间片循环完成，但我们也不可否认，这种改写成本很高。如果我们把程序逻辑中所有的循环全部以新的写法替代，势必造成程序过于冗长，繁琐，程序员疲于奔命，导致新的 bug 危机。

因此，这里面有一个很重要的“度”的把握，即我们使用任务池来替换的线程循环，仅限于那些纯“守候循环”，即无意义地等待某个状态的改变，某个事件的发生的循环，而对于正常的业务计算逻辑中的循环，如一个数组的遍历，一次业务的查询，这类业务循环，我们还是保留，以实现开发效率和运行效率的平衡。

这么说可能有点抽象，我们举一个例子。比如说一个网络服务器，它有很多守候线程，在监听很多个客户的 socket 连接。当一个客户通过一个 socket，发出一个请求，请求查询属于陕西省西安市的所有的男性用户名单（我们假定这个用户名单在内存里），这个时候，势必引起遍历查询的循环。

那我们来界定一下，这个查询循环，我们不做任务池优化，该怎么写还怎么写，但是，前面的守候线程，我们应该优化，把它改写成“守候任务”，即改成任务池中的任务，每次运行，查询一次，然后返回，等待下次运行。

因此，我们这里明确界定任务池任务的粒度，任务池的“任务”，通常只原线程池的守候线程的核心逻辑，即守候、等待某一事件发生，某一状态改变，并引发动作的这部分逻辑，不包含常规业务计算逻辑的循环。

9.4.2.2 任务池的边界考虑

我们设计任何一个功能模块前，一般要考虑其边界，这是常规思维。那么，何谓“边界”呢？

一般说来，一个程序模块的“边界”，是指其数据边界，即其最大、最小处理的数据范围，其处理的数据粒度等，但笔者认为，边界设计，还有个最重要的功能，即说明本程序模块，能做什么，不能做什么。

任务池也一样，虽然我们完全可以从操作系统层面，引申出线程应用，再进一步完善任务链表，最终实现任务逻辑，但是笔者并不打算这么做，原因是太复杂了。任务池的核心功能目的，就是把线程任务细化，切片成单一、可重入的动作任务，而如何与操作系统进行线程沟通，这部分一来线程池已经实现，没必要再次重复工作，二来，这也不是任务池的主要功能目的。

笔者经过思考，界定任务池就是在线程池的基础上，做进一步的功能细分，实现任务切片，所有线程相关操作，全部由原线程池完成。换言之，任务池是线程池的使用者，而不是替代者。

9.4.2.3 任务池实施原理

可能各位读者已经关注到，任务池在实施时，有一个技术难点，就是任务链表如何实现，以及如何实现多个任务线程，和谐地从任务链表中取得任务执行，既不能重复执行，也不能遗漏。

这里我们可能有一个误区，由于前文为了描述方便，尤其是笔者需要明确地提示大家，操作系统的任务链表，是动态的，是随时可以加入和删除任务的，因此，使用了“链表”这个概念。

但笔者在实际构造任务队列是，发现其实没有必要用“链表”来实现，我们来看看任务池的设计要求：

- 1、要有一个数据中心，数据中心里面的数据，描述各个任务片段。
- 2、数据中心内部的数据必须有一定顺序性，不能乱序，以保证各个任务片段，能被公平地选中，激活运行。
- 3、一个任务片段，在数据中心内时，可以无差别被任何一个执行线程选中执行，一旦被选中，在执行期间，其不应该存在于数据中心，以免被重复选中，double 执行。
- 4、当一个任务片段被执行完毕后，应该放到序列的末尾，不能被以压栈方式放在序列头，否则，其下次会第一个被再次选中，其优先权就远高于其他兄弟任务，这不合理。

这就是任务池数据中心的需求，我们来看看，有顺序，从序列头提取任务，执行完毕放回序列尾，嗯，先进先出逻辑，这就是队列嘛。我们用图来看看是不是这么回事：




图 9.19: 线程执行体 1, 从任务队列头, 弹出任务 1, 准备执行




图 9.20: 线程执行体 1 执行任务 1, 线程执行体 2, 从任务队列头, 弹出任务 2, 准备执行




图 9.21: 线程执行体 2 执行任务 2, 线程执行体 1 执行完任务 1 片段, 将其推入任务队列末尾, 同时, 从队列头弹出任务 3, 准备执行

如上述几个图所示, 所有的线程执行体, 不断从任务队列头, 弹出任务, 执行一个片段, 然后将该任务描述推回到队列末尾, 以此类推, 不断循环。实现整个任务池的工作逻辑。

这里有几个要点, 请大家关注:

1、当任务从队列头弹出后（请注意是弹出），队列中不再有这个任务的描述，因此，任务不会被重复选中，double 运行。

2、任务队列与线程执行体分离，换言之，有多少线程并不重要，无非是线程多，任务队列“翻动”就快，看起来执行效率就高，而线程少，“翻动”慢，执行效率低，但这已经不是绝对影响效率的因素了

3、一个任务，本次可能被线程 1 选中执行，下次可能被线程 2 选中执行，都无所谓。由于高度的抽象性，线程提供标准的时间片执行资源，任务池提供标准描述的任务数据，因此，任务和线程没有必然绑定性，这导致了最大的执行灵活性。换言之，即使某个线程被某个长程任务挂住，不能及时返回，但由于任务池的其他线程还在高速“翻动”任务队列，因此，对整个任务池的效率影响很小。

4、任务队列可以很长，甚至达到几万个任务，但线程数可以恒定，这说明我们不再受到原有操作系统的线程数限制，并发的任务数可以达到几万个。

5、当某个任务执行完毕，由于是任务数据是从任务队列中“弹出”，因此，线程只需要“不”将其推回队列，即自然实现了删除任务动作。

以上就是利用队列来实现任务池的核心设计原理。

这么设计还有个最大的好处，就是当某个时刻由于业务上的原因，突然并发任务量剧增时，表现到任务池上，仅仅是任务队列有所增加，而不是操作系统的线程剧增，这大大缓和了峰值业务压力下，对操作系统的冲击，使软件系统的安全性得到了极大的提高。这在商用数据传输工程的服务器开发中，是至关重要的特性。

9.4.2.4 任务池设计方案

当我们想通了任务池的核心实施原理，那么，其设计方案也呼之欲出了。经过考虑，笔者决定按如下逻辑设计任务池：

1、任务池为一个 C++ 标准类，依赖线程池、以及前文的 MemQueue 工作，以保证高效的工作效率。

2、任务池内聚一个任务队列，并且，开启在线程池中开启有限数量的任务线程备用。

3、任务池提供注册方法，允许其他模块将任务注入到任务队列中。

4、任务池内的线程，以守候方式，不断检索任务队列，有任务则执行一次，将任务推回任务队列，然后睡眠。

9.4.3 任务池的基本定义说明

任务池最重要的定义，应该是其回调函数构型，和线程池有了较大的差异性。我们先来看看线程池的回调函数构型：

```
typedef void (*_TPOOL_CALLBACK) (  
    void* pCallParam,           //这是主程序透传进来的参数指针  
    MBOOL& bThreadContinue); //如果系统要退出,线程池会修改这个参数的值为 false
```

这里我们关注到，线程池的回调函数，除了替应用层透传一根 void* 的参数指针之外，还传递了一个 MBOOL& bThreadContinue 变量。这在前面的介绍中，我们已经能理解其含义了。

由于线程池内的线程，还是类似于单任务操作系统的工作方式，一旦执行一个用户线程回调函数，就是把所有的操作权交给该函数代码，自己丧失了控制权，因此，对于内部包含死循环的守候线程任务，必须传递这么一个 bool 类型的线程安全变量，并且强迫应用线程的死循环要写成如下格式：

```
while (XMG(bThreadContinue))           //这是标准写法，即线程池退出支持
```

以此来保证线程池退出时，可以强迫所有的子任务退出，避免由于丧失控制权，导致的退出挂死现象。

而任务池由于我们设计时，已经取缔了这类守候循环，因此，没有必要再传递这个 `bThreadContinue` 参数，反过来，我们看看前文的例子，有一个重要参数需要引起我们的关注：

```
void AppFunc(void)
{
    //nStatus 是最关键的，函数状态机变量，记录了函数目前的运行状态
    //注意，这是一个静态变量，函数退出，变量值还存在，不会被摧毁
    //当然，也可以使用类成员变量，或者全局变量等生命周期长于函数本身的变量代替。
    static nStatus=0;
    switch(nStatus)                //这就是执行机，根据状态机，决定运行的程序段落
    {
        //...
```

大家注意，当我们把守候循环改写成可重入的代码时，需要一个很重要的状态机 `nStatus`，由于明确要求这个 `nStatus` 的生命周期要比函数体长，以长期保存状态，因此，任务池有必要为每一个任务，保存这个状态值，并允许任务函数在其中修改这个状态值，改变自己的行为。

因此，笔者经过考虑，把任务池的回调函数定义为如下构型：

```
//任务池回调函数构型
//返回值 true 继续运行 false 停止运行
typedef bool (*_TASKPOOL_CALLBACK)(void* pCallParam,    //代传参数指针
                                   int& nStatus);       //程序控制状态机
```

首先，我们需要确定，任务本身有权利停止自己，因此，回调函数设置返回类型为 `bool` 量，当应用层任务返回 `false`，表示该任务已经结束，任务执行线程就不会将其重新送回任务队列，而是直接抛弃，这个任务也就真实地结束了。

其次，我们注意到，参数中以传址调用传递了一个状态机参量，这是任务池代每个任务保管的状态机服务，任务回调函数可以用，也可以不用。

当回调函数构型确定以后，我们还需要确定两个细节，如下所示：

```
#define TASK_POOL_TOKEN_MAX (16*1024) //最多同时并发任务数
#define DEFAULT_THREAD_MAX (30)      //默认线程数
```

请注意这里，这两个宏定义，前者定义出任务池最多的并发任务数，通常是 16384 个任务，另外一个默认的线程数，笔者这里默认是 30 条。这些参量，各个读者可以根据具体情况调整。

这里需要特别说明一点，任务池的并发执行效率，已经和线程池没有太大的关系，由于任务池中，一启动就是 30 条线程并发，原则上，只要任务队列中有足够多的任务，这三十条线程总是有事情做，线程时间片在各个任务间高度重用，因此，其效率甚至比原有各自为政的线程池中 300 条线程的效率都高，任务池的最大默认线程数与执行效率之间，不是线性关系。

根据笔者经验，并发 10000 个左右的任务，大约不超过 100 条线程，足以应对，且每个任务的执行效率都不低。

当然，最后我们还是介绍一下任务池的核心数据结构，任务描述结构，这是任务池最重要的数据结构，充分体现的任务池的执行逻辑。

```
//任务池核心数据结构
typedef struct _TASK_POOL_TOKEN_
{
    _TASKPOOL_CALLBACK m_pCallback;    //回调函数指针
    void* m_pUserParam;                //替用户传递的回调函数参数，可以为 null
    int m_nUserStatus;                 //代替用户传递的一个状态值，初始值默认是 0
}STaskPoolToken;
//数据结构长度
const int STaskPoolTokenSize=sizeof(STaskPoolToken);
```

大家可能注意，针对每个任务，任务池在数据结构中，提供了状态机服务，每次任务注册，状态机默认从 0 开始计数，并允许任务回调在运行中动态修改。

9.4.4 任务池的类声明

任务池本质上还是一个高级的应用类，它综合应用了前文所述线程池、队列、内存池、锁等各方面的知识，因此，它对上述模块都有依赖性。在下面的类生命中，有一个类尚未对读者介绍，就是 **CTonyBaseLibrary**。这是笔者的一个聚合类，目的就是把所有的工具类聚合到一个类中统一管理，方便使用，这在下文有专门的章节介绍，这里读者先了解一下即可。


```

//依赖类
class CTonyBaseLibrary;
class CTonyThreadPool;
class CTonyMemoryQueueWithLock;
class CTonyMemoryPoolWithLock;
class CTonyLowDebug;
//任务池类
class CTonyXiaoTaskPool
{
public:
    CTonyXiaoTaskPool(
        CTonyBaseLibrary* pTonyBaseLib,           //依赖的工具聚合类指针
        int nMaxThread=DEFAULT_THREAD_MAX);       //最大线程数
    ~CTonyXiaoTaskPool();
public:
    bool ICanWork(void);                          //防御性设计，可运行标志
    void PrintInfo(void);                         //内容打印，Debug 用
public:
    //注册一个新任务，返回 TaskID
    bool RegisterATask(_TASKPOOL_CALLBACK pCallback, //回调函数指针
        void* pUserParam=null);                   //回调函数参数
private:
    //真实的内部注册函数
    bool RegisterATaskDoIt(STaskPoolToken* pToken,int nLimit=-1);
    //服务器线程
    bool TaskServiceThreadDoIt(STaskPoolToken& Task);
    static void TaskServiceThread(void* pCallParam,
        MBOOL& bThreadContinue);
    //管理者线程
    static void TaskCtrlThread(void* pCallParam,
        MBOOL& bThreadContinue);
private:
    CMRSWbool m_bThreadContinue;                  //任务池自带线程管理变量
    CMRSWint m_nThreadCount;                      //可以自行退出所属线程
    CTonyMemoryQueueWithLock* m_pTaskQueue;       //核心任务队列
    CTonyThreadPool* m_pThreadPool;               //线程池指针
private:
    int m_nMaxThread;                             //最大任务数的保存变量
    CMRSWint m_nThreadID;                         //任务 ID 种子
    CTonyLowDebug* m_pDebug;                     //debug 对象指针
    CTonyBaseLibrary* m_pTonyBaseLib;            //聚合工具类指针
};

```

9.4.5 构造函数和析构函数

任务池的构造函数和析构函数相对比较简单，这是由于任务池内部也实现了一个管理者线程，真实的服务线程启动，是由管理者线程完成的。这里是构造函数的介绍：

```

CTonyXiaoTaskPool::CTonyXiaoTaskPool(CTonyBaseLibrary* pTonyBaseLib,
                                       int nMaxThread)
{
    m_pTonyBaseLib=pTonyBaseLib;           //保存聚合工具类指针
    m_pDebug=m_pTonyBaseLib->m_pDebug;     //从聚合工具类中获得 debug 对象指针,
                                           //方便后续使用
    m_nMaxThread=nMaxThread;               //保存最大线程数
    XGSyslog("CTonyXiaoTaskPool(): Start!\n"); //打印启动信息
    m_bThreadContinue.Set(true);           //线程管理变量初始化
    m_nThreadCount.Set(0);
    m_nThreadID.Set(0);                    //任务 ID 种子赋初值 0
    //实例化任务队列
    m_pTaskQueue=new CTonyMemoryQueueWithLock(
        m_pTonyBaseLib->m_pDebug,           //传入 debug 对象指针
        m_pTonyBaseLib->m_pMemPool,         //传入内存池指针
        "Tony.XiaoTaskPool");              //应用名
    if(m_pTaskQueue)
    {   //注册到内存池进行管理, 预防退出时忘了释放
        TONY_REGISTER(m_pTaskQueue, "CTonyXiaoTaskPool::m_pTaskQueue");
    }
    //实例化线程池
    m_pThreadPool=new CTonyThreadPool(m_pTonyBaseLib->m_pDebug);
    if(m_pThreadPool)
    {   //注册到内存池进行管理
        TONY_REGISTER(m_pThreadPool, "CTonyXiaoTaskPool::m_pThreadPool");
    }
    if(ICanWork()) //判断前面的 new 动作是否完成
    {   //启动管理线程
        if(!m_pThreadPool->ThreadPoolRegTask(TaskCtrlThread, this))
        {   //这是日志输出, 后面章节有详细介绍
            XGSyslog("CTonyXiaoTaskPool:: start ctrl thread %d fail!\n");
        }
        else
            m_nThreadCount.Add(); //如果注册成功, 线程计数器+1
    }
}

```

这里是析构函数的实例。

```

CTonyXiaoTaskPool::~~CTonyXiaoTaskPool()
{
    //以标准方式退出本对象所属所有线程
    m_bThreadContinue.Set(false);
    while(m_nThreadCount.Get()){Sleep(100);}
    TONY_DEL_OBJ(m_pThreadPool);           //删除对象的宏
    TONY_DEL_OBJ(m_pTaskQueue);
    XGSyslog("CTonyXiaoTaskPool(): Stop!\n"); //打印退出信息
}

```

这里面有几个宏, 还没有为大家介绍, 就是 TONY_DEL_OBJ, 其定义如下:

```
//将一根指针注册到内存池，内存池默认在聚合工具类中
#define TONY_REGISTER(pPoint,szInfo) \
    m_pTonyBaseLib->m_pMemPool->Register(pPoint,szInfo)
//将一根指针从内存池反注册，内存池默认在聚合工具类中
#define TONY_UNREGISTER(pPoint) \
    m_pTonyBaseLib->m_pMemPool->UnRegister(pPoint)
//将一根对象指针先执行反注册，再摧毁，最后置空
#define TONY_DEL_OBJ(p) \
    if(p){TONY_UNREGISTER(p);delete p;p=null;}
```

这里，我们顺便也介绍一下任务池的工具函数。

```
bool CTonyXiaoTaskPool::ICanWork(void)
{
    //主要检查任务队列和线程池对象是否初始化
    if(!m_pThreadPool) return false;
    if(!m_pTaskQueue) return false;
    return true;
}

void CTonyXiaoTaskPool::PrintInfo(void)
{
    //打印当前线程数和任务书，方便程序员观察
    TONY_XIAO_PRINTF("task pool: thread count=%d, task in queue=%d\n",
        m_nThreadCount.Get(),m_pTaskQueue->GetTokenCount());
}
```

9.4.6 管理者线程

任务池的管理者线程逻辑非常简单，就是根据总的线程数，启动对应数量的服务线程，实现后续的任务机执行。当所有服务线程启动完毕，管理者线程退出。因此，任务池的服务线程启动过程，相对主线程来说，是异步的。请注意，“任务池”的管理者线程和服务者线程，实际上是“线程池”的线程回调函数。

```
void CTonyXiaoTaskPool::TaskCtrlThread(
    void* pCallParam,           //这是主程序传进来的参数指针
    MBOOL& bThreadContinue)     //如果系统要退出，线程池会修改这个参数的值为 false
{
    CTonyXiaoTaskPool* pThis=(CTonyXiaoTaskPool*)pCallParam;
    int i=0;
    for(i=0;i<pThis->m_nMaxThread;i++) //根据最大线程数，循环启动服务线程
    {
        if(!pThis->m_pThreadPool->ThreadPoolRegTask(
            TaskServiceThread,pThis))
        {
            //启动失败，打印报警信息，退出启动
            pThis->XGSyslog("CTonyXiaoTaskPool:: start service \
thread %d fail!\n",i);
            break;
        }
        else
            pThis->m_nThreadCount.Add(); //启动成功，线程数+1
    }
    pThis->m_nThreadCount.Dec(); //任务完成，自己退出，因此线程计数器-1
}
```

这里有个很重要的细节需要和大家沟通，笔者在这里设计管理者线程的目的，至少有一个原因是希望在以后需要的时候，可以实现任务池线程的动态起停，也就是说，在以后

开发需要时，笔者可能会改写这段函数，使任务池的服务线程数也成为可变的，这样可以进一步优化系统资源占用。

不过，起码目前笔者面临的开发环境，还不需要这种深入的设计，因此，笔者在这里留出了这个线程函数作为设计接口。以后的动态负载均衡，将在此处实现。

9.4.7 服务者线程

任务池的服务者线程和线程池的同名函数有了较大差异，其核心关键就是此处的服务者线程，就是守候线程，它总是守候等待任务队列中的任务，有任务，则执行，无任务，则睡眠等待。

为了简化设计，按照 C 和 C++ 无错化程序设计方法中的原则，笔者把服务者线程分为两个函数，线程体函数和任务执行函数。

任务执行函数主要实现具体的任务执行逻辑，即“从任务队列提取---执行---推回任务队列”的逻辑。

```
//传入任务数据结构 STaskPoolToken 指针，
//根据任务回调情况，返回真或者假，返回假，表示该任务已经结束
bool CTonyXiaoTaskPool::TaskServiceThreadDoIt(STaskPoolToken& Task)
{
    bool bCallbackRet=                //执行回调函数，并获得返回值
        Task.m_pCallback(Task.m_pUserParam,Task.m_nUserStatus);
    if(!bCallbackRet) return bCallbackRet; //如果返回值为假，直接返回，表示任务结束
    //如果任务返回真，表示任务尚未结束，需要重新推回任务队列，等待下次运行
    bCallbackRet=RegisterATaskDoIt(&Task); //试图重新注册到任务队列
    if(!bCallbackRet)                    //如果注册失败，报警，表示任务丢失
    {
        TONY_DEBUG("CTonyXiaoTaskPool::TaskServiceThreadDoIt(): a task need
continue, but add 2 queue fail! task lost!\n");
        TONY_DEBUG_BIN((char*)&Task,STaskPoolTokenSize);
    }
    return bCallbackRet;                //返回最后结果
}
```

线程体函数主要负责和线程池的交互，维护线程池需要的回调线程逻辑完整性，构建一个完整的守候循环。

```

void CTonyXiaoTaskPool::TaskServiceThread(
    void* pCallParam,          //这是主程序传进来的参数指针
    MBOOL& bThreadContinue)    //如果系统要退出,线程池会修改这个参数的值为 false
{
    int nQueueRet=0;
    STaskPoolToken Task;
    char* szTask=(char*)&Task;
    //获得本对象指针
    CTonyXiaoTaskPool* pThis=(CTonyXiaoTaskPool*)pCallParam;
    int nID=pThis->m_nThreadID.Add()-1;
    //这是打印本服务线程启动信号,通常注释掉不用,调试时可能使用。
    //pThis->XGSyslog("CTonyXiaoTaskPool::TaskServiceThread():          %d
Start!\n",nID);
    while(XMG(bThreadContinue))    //注意本循环,标准的线程池线程守候循环
    {    //额外增加的判断,以便判断本任务池退出标志
        if(!pThis->m_bThreadContinue.Get())
            goto CTonyXiaoTaskPool_TaskServiceThread_End_Process;
        //尝试从任务队列中弹出第一个任务,并执行
        nQueueRet=pThis->m_pTaskQueue->GetAndDeleteFirst(
            szTask,STaskPoolTokenSize);
        if(STaskPoolTokenSize==nQueueRet)    //如果弹出成功,表示有任务
        {
            pThis->TaskServiceThreadDoIt(Task);    //调用上一函数执行
        }
        Sleep(MIN_SLEEP);    //习惯性睡眠
    }
CTonyXiaoTaskPool_TaskServiceThread_End_Process:
    pThis->m_nThreadCount.Dec();    //退出时,线程计数器-1
    //这是打印本服务线程停止信号,通常注释掉不用,调试时可能使用。
    //pThis->XGSyslog("CTonyXiaoTaskPool::TaskServiceThread():          %d
Stop!\n",nID);
}

```

9.4.4 任务注册接口

任务池最重要的接口函数,也可以说是唯一的公有业务接口,就是任务注册接口。但是这里有点小小的差别,我们知道,任务池的任务队列注册,有两个需求调用,一个是任务服务线程完成一个任务后,如果任务没有结束,需要注册回任务队列,等待下次运行,另一个就是公有的,业务层需要注册新任务了。因此,这个函数也有两个构型。

第一个构型,主要应对内部的注册需求,即服务线程的任务注册,由于服务线程掌握的是任务描述数据结构 STaskPoolToken,因此,这个函数的参数也是这个结构体指针。

```

bool CTonyXiaoTaskPool::RegisterATaskDoIt(STaskPoolToken* pToken,int nLimit)
{
    bool bRet=false;
    if(STaskPoolTokenSize==m_pTaskQueue->AddLast(    //一次典型的队列 AddLast 行为
        (char*)pToken,STaskPoolTokenSize),nLimit)
        bRet=true;
    return bRet;
}

```

而考虑到外部业务层的注册行为，并不理解这个关键的数据结构体 `STaskPoolToken`，因此笔者又准备了一个专门应对外部请求的注册函数。

```
bool CTonyXiaoTaskPool::RegisterATask(
    _TASKPOOL_CALLBACK pCallback,           //传入回调函数指针
    void* pUserParam)                       //传入参数指针
{
    STaskPoolToken Token;                   //准备一个结构体实例
    if(!ICanWork()) return false;
    if(!pCallback) return false;
    Token.m_pCallback=pCallback;            //填充结构体
    Token.m_pUserParam=pUserParam;
    Token.m_nUserStatus=0;                  //注意此处，状态机置为 0
    return RegisterATaskDoIt(&Token,m_nMaxThread); //调用上面的函数，完成注册
}
```

9.4.4 任务池的小结及实现示例

看完上面的介绍，可能大家会有一个奇怪的想法，这个任务池很简单，甚至简单到超出想象。笔者其实也有这个感觉。

笔者书写任务池时，大约只用了三个小时左右，就完成了上述代码，并且测试一次通过。以后也再也没有出现过问题。但是，笔者在产生任务池这个抽象逻辑思维时，正如上面的介绍一样，非常复杂，笔者大约用了一年的时间才把整个任务池的逻辑思考完整。

这大概是程序设计的通理了，一个具有强大功能的逻辑，一旦想透，实施起来，可能非常简单。但是，后来的程序员，要想从这段简单的代码中，还原出原始的深邃逻辑思维，其实是一种不太可能的问题。

因此笔者在本书中，特别强调知识的传承性，强调知识的实用性，并遵循先讲理论，再展示实例，理论与实践相结合的原则，就是为了让各位读者不但理解原理，也懂得应用，最终能把知识应用到工程实战中，真实有效地帮助大家完成工作任务。希望各位读者深刻理解这一点。

这里，笔者给大家展示一段几乎可以称之为标准的任务池任务回调函数示例，大家可以体会一下可重入任务的概念。

```

//任务池任务函数示例，静态函数
static bool TaskCallback(void* pCallParam,int& nStatus)
{
    //...
    switch(nStatus)
    {
    case 0:      //init      //初始化代码
        //...
        nStatus++;
        break;
    case 1:      //loop      //循环体
        //...
        if(...)      //某种条件下，中断循环
            nStatus++;
        break;
    default:     //exit      //结束代码
        //...
        return false;      //返回假，任务结束
    }
    return true;      //正常情况下，返回真，任务继续
}

```

9.5 任务池的运行体

在任务池完成后，笔者立即将其应用到工程中，一面调试，一面也直接用于解决问题。但在笔者工作中，发现了一个问题：“任务池不方便”。

原因很简单，看上文的例子，每个任务都要写成上述模型，里面建立一个庞大的 switch 语句，这显得很繁琐。笔者的习惯是每段代码，每段逻辑，只写一次。一来减少工作量，二来也减少 bug 的可能。但任务池规定的这种函数写法，显然不符合这个原则。

因此，笔者经过思考，为了简化任务池的调用模型，又设计了任务池运行体 CTaskPoolRun 这个模块，专门用于简化应用层的开发。

但同任务池相对线程池的抽象一样，这个运行体 CTaskPoolRun 相对任务池，也是进一步抽象思维的结果，简单的设计下面，也包含了较为深层次的思考。

9.5.1 简化运行态

其实从上文的例子我们已经可以看出，虽然我们的任务池提供了状态机服务，并且，每个任务回调函数内部，也采用了 switch 结构，但实际工程中笔者发现，绝大多数时候，不管我们承不承认，其实状态机就只有三种状态：

- 1、初始化代码
- 2、循环体代码
- 3、结束代码

因此，上述复杂的 switch 结构，其实完全可以简化为三段函数，TaskInit，TaskLoop 和 TaskExit，如果我们能把每个任务进一步抽象为这三段函数的组合，则我们可以进一步简化任务函数的开发，回避掉复杂的 switch 结构。

因此，笔者就试图在任务池的基础上，进一步封装一层较为方便的调用，即任务运行体 CTaskPoolRun。代应用层实现复杂的 switch 调用，而让应用层程序员（其实就是笔者自己），更加专注于应用层业务逻辑，不要在任务、线程的调用细节，浪费太多的精力。

9.5.2 任务描述工具类

虽然笔者这轮开发的目的，是设计一种较为方面的任务调用模型，以简化应用层任务设计。但是，笔者在开发任务运行体 CTaskPoolRun，还需要首先回答一个问题，即怎么注册任务才最方便。

经过思考，笔者开发了一个纯工具类 CTonyTaskRunInfo，这是典型的工具类，目的就是尽量方便地帮助应用层描述任务，进而注册到任务运行体中运行。

9.5.2.1 任务描述工具类需求分析

在构建这个类时，笔者思考了一个问题，如何才能提供“最大的使用方便性”。使用方便的类，无非是笔者前文提出的“工具类”和“粘合类”，工具类，就是内部所有核心数据公开，更像一个提供了方便的访问方法的结构体。粘合类就更进一步，连核心数据都是借用别人的，完全是一些方法的集合，贴合到某个数据结构上方便操作。

经过考虑了，笔者决定，CTonyTaskRunInfo 把这两种特性都实现出来，即既是工具类，也是粘合类。同时，大量使用重载，尽可能方便用户调用。

经过考虑，笔者准备以如下机制设计 CTonyTaskRunInfo：

1、内部包含一个数组，即可以容纳多个任务回调函数，当然也可以只有一个，这样，CTonyTaskRunInfo 既可以描述单一回调任务，也可以描述多笔，有关联关系的回调业务。

2、提供 AddLast 方法，供应用层添加回调任务，添加顺序默认就是执行顺序，不再设置 Index

3、每个 CTonyTaskRunInfo 描述相关联的一组回调任务，因此，所有的回调函数共用一根 void* 的参数指针

4、最多设置 16 步回调，即数组最大长度 16，根据经验，这已经足够了。

5、同时实现工具类和粘合类，关键函数多种重载，保持最大使用方便性。同时，为了保持高效性，尽量使用内联函数完成开发。

9.5.2.2 核心数据结构

在设计 CTonyTaskRunInfo 之前，我们必须设计其核心数据管理结构。

```
#define TONY_TASK_RUN_MAX_TASK 16 //最多步动作
typedef struct _TONY_TASK_RUN_INFO_
{
    int m_nTaskCount;           //总共多少步骤
    void* m_pCallParam;         //共用的回调函数参数
    //动作回调函数数组
    _TASKPOOL_CALLBACK m_CallbackArray[TONY_TASK_RUN_MAX_TASK];
} STonyTaskRunInfo;
const ULONG STonyTaskRunInfoSize=sizeof(STonyTaskRunInfo);
```

另外，根据后文中 CTonyTaskRun 的具体使用需求，笔者还设置了一个信息较为完整的任务描述结构，此处先做简要介绍，后文会有更加详细的说明。


```

class CTonyTaskRun;
typedef struct _TonyTeskRunTaskCallback_Param_
{
    STonyTaskRunInfo m_Info;                //任务描述结构体
    CTonyBaseLibrary* m_pTonyBaseLib;       //基本聚合工具类指针
    CTonyTaskRun* m_pThis;                  //任务运行体对象指针
    int m_nRunIndex;                        //当前执行的步距
    char m_szAppName[TONY_MEMORY_BLOCK_INFO_MAX_SIZE]; //应用名
}STonyTeskRunTaskCallbackParam;
const ULONG STonyTeskRunTaskCallbackParamSize= //结构体长度常量
    sizeof(STonyTeskRunTaskCallbackParam);

```

9.5.2.3 内部核心数据

由于笔者的目标，是将本类既实现成工具类，也实现成粘合类，因此，内部实际上有两个数据区。

```

public:
    STonyTaskRunInfo m_Info;                //作为工具类的数据区实体，保存数据
    STonyTaskRunInfo* m_pInfo;              //作为粘合类的数据区指针，指向外部数据

```

笔者这么预设，如果是工具类方式启动，则将 **m_pInfo** 置为 **null**，如果是粘合类启动，则 **m_pInfo** 被赋给外部传入的参数区指针。所有的相关函数，执行时先检查 **m_pInfo** 的值，如果不为 **null**，则操作 **m_pInfo**，否则操作 **m_Info**。这样就能一套代码，实现两种操作。

9.5.2.4 init 工具函数

本类是一个完全内联的类，即其中所有方法均在类声明中以 C++ 的内联方式直接完成。以下所有代码都是直接写在类声明中。

init 这个函数本质上与对象无关，因此，笔者设计为静态函数，数据区的指针需要参数传入。

```

private:
    static void Init(STonyTaskRunInfo* pInfo) //初始化动作
    {
        pInfo->m_nTaskCount=0;                //动作计数器归零
        pInfo->m_pCallParam=null;              //参数指针清空
        int i=0;
        for(i=0;i<TONY_TASK_RUN_MAX_TASK;i++) //清空 16 个回调函数指针
            pInfo->m_CallbackArray[i]=null;
    }

```

9.5.2.5 GetPointInfo

由于内部可能存在两个数据区，因此，需要设置一个公有方法，获得当前实际使用的数据区。

```

public:
    STonyTaskRunInfo* GetInfoPoint(void)
    {
        if(m_pInfo)                //优先返回 m_pInfo
            return m_pInfo;
        else
            return &m_Info;         //否则返回 m_Info 的地址
    }

```

9.5.2.6 构造函数和析构函数

本类为了方便，大量重载了构造函数，使之尽量适应各种使用情况。

```
public:
    //这是使用 STonyTeskRunTaskCallbackParam 结构传参，实现粘合的构造函数
    //使用 m_pInfo 指针，指针指向 STonyTeskRunTaskCallbackParam 中的 m_Info
    CTonyTaskRunInfo(STonyTeskRunTaskCallbackParam* pParam)
    {
        m_pInfo=&(pParam->m_Info);
        Init(m_pInfo); //初始化
    }
    //这是直接粘合到外部的一个 STonyTaskRunInfo 结构体实例的重载构造函数
    //使用 m_pInfo 指针，指针指向传入的结构体指针
    CTonyTaskRunInfo(STonyTaskRunInfo* pInfo)
    {
        m_pInfo=pInfo;
        Init(m_pInfo); //初始化
    }
    //这是以工具类启动，不使用 m_pInfo，而使用本类自带的数据区
    CTonyTaskRunInfo()
    {
        m_pInfo=null;
        Init(&m_Info); //初始化
    }
    ~CTonyTaskRunInfo(){} //析构函数不摧毁数据
```

9.5.2.7 SetCallbackParam

由于本类仅仅包含相关联的一组回调任务，因此，定义所有的回调函数，都使用相同的参数指针。本函数设置共用参数指针。

```
public:
    void SetCallbackParam(void* pCallParam)
    {
        if(m_pInfo) //注意先后顺序，优先使用 m_pInfo
            m_pInfo->m_pCallParam=pCallParam;
        else
            m_Info.m_pCallParam=pCallParam;
    }
```

9.5.2.8 AddTask

AddTask 是最关键的一个函数，主要是以队列方式，将新的任务回调函数，添加到内部数据区的数组末尾。

```
public:
    //这是添加一个回调函数，连同其参数的方法，调用了下面的单独回调函数添加方法
    bool AddTask(_TASKPOOL_CALLBACK pCallback,void* pCallParam)
    {
        if(pCallParam) SetCallbackParam(pCallParam); //有参数，则设置参数
        return AddTask(pCallback); //调用下一函数，处理回调指针
    }
```

这是真实实现回调函数指针队列的追加功能。请注意，如果队列满了（任务回调函数总数超过 16 个，返回 false）

```

bool AddTask(_TASKPOOL_CALLBACK pCallback)
{
    if(m_pInfo)                //优先使用 m_pInfo
    {    //检查回调函数总数是否超限，是则返回 false，拒绝添加
        if(TONY_TASK_RUN_MAX_TASK<=m_pInfo->m_nTaskCount) return false;
        //添加到数组末尾
        m_pInfo->m_CallbackArray[m_pInfo->m_nTaskCount]=pCallback;
        //数组计数器+1
        m_pInfo->m_nTaskCount++;
        return true;
    }
    else                        //无 m_pInfo 可用，则使用 m_Info
    {    //检查回调函数总数是否超限，是则返回 false，拒绝添加
        if(TONY_TASK_RUN_MAX_TASK<=m_Info.m_nTaskCount) return false;
        //添加到数组末尾
        m_Info.m_CallbackArray[m_Info.m_nTaskCount]=pCallback;
        //数组计数器+1
        m_Info.m_nTaskCount++;
        return true;
    }
}

```

9.5.2.9 CopyFrom

考虑到 STonyTaskRunInfo 是复杂数据结构，因此此处单设了一个拷贝工具函数，方便应用层以后的使用。本函数顾名思义，将给出的一个数据结构体的内容，拷贝到当前本对象使用的数据结构体中（m_pInfo 或者 m_Info）

```

public:
    void CopyFrom(STonyTaskRunInfo* pInfo)
    {
        char* pMyInfo=null;                //查找有效的拷贝目标对象
        if(m_pInfo) pMyInfo=(char*)m_pInfo;
        else pMyInfo=(char*)&m_Info;
        //二进制格式拷贝
        memcpy(pMyInfo, (char*)pInfo, STonyTaskRunInfoSize);
    }

```

9.5.2.9 CTonyTaskRunInfo 小结

各位读者看完前面的代码，可能会有各种感觉，至少，很多读者可能会觉得很繁琐，很累赘，程序不简洁，甚至不符合 C 和 C++ 无错化程序设计思想。一个简单的工具类，甚至没有它也无所谓，至于考虑这么多吗？

这里笔者特别需要给读者一点程序开发思维的提示：

1、笔者很看重这类工具类，因为开发中，很多时候的 bug，并不是程序员喜欢写错误的代码，很多时候，就是忘了，或者没有关注到某个细节，又或者，某个模块的调用接口太复杂，没有完全理解，就贸然写程序，导致错误。但最困难的是，这种情况很难通过程序员个人调整去杜绝。

2、这类完全为了方便使用的工具类，笔者很少看到程序员有意识地书写，但笔者自己有这个习惯，目的很简单，磨刀不误砍柴工，开发前，先用点时间，做点准备工作，好过后期修改无数的 bug。

3、任务池总的来说，还是一个很复杂的概念，要想用好，还是需要程序员细心地开发，但这种细心，是不能保证的，程序员很可能因为某一天的熬夜，脑子一昏，第二天就写错了代码，并把这个错误一直带到最后来改 bug。那为什么我们不能写出一个方便的调用工具，保证程序员写不错呢？笔者正是基于这一思维，才开发了 CTaskPoolRun 和 CTonyTaskRunInfo 这两个类。出发点很简单，现在笔者脑子清醒，就借着这股清醒劲，写出一个绝对正确的调用模型，以后就再也不用每次都动脑子想了，免得想错。

4、工具类往往很繁琐，因为工具类一般没有严密的设计，都是先搭个框子，以后需要哪个功能，添加哪个功能，代码是在不断完善的，可能大家觉得代码不够严密，但是，经过笔者试验，上述代码在实战中，确实好用。

5、由于没有严密的设计，因此，工具类的函数方法，原则上应该写得够简单，杜绝一些很复杂的算法，笔者没有在 CTonyTaskRunInfo 使用内存队列等复杂数据结构，就是简简单单一个数组，为什么？程序看起来一目了然，想错都很难。

6、工具类永远只提供有限服务，我们可以注意到 AddTask 函数，原则上它执行的是队列逻辑，但是，队列满了怎么办？满了就满了嘛，如果一个任务真的需要超过 16 步任务回调，那扩展 TONY_TASK_RUN_MAX_TASK 这个宏，把 16 改大就好了，无非多占点内存，但笔者绝对不会再去加复杂的逻辑自动处理抛弃等逻辑，因为不划算。因此，AddTask 的 false，很多时候，是返回给程序员看的，程序员看到了，就去把 16 加大，而不是写程序处理，总之，发布版，是不允许 AddTask 出现 false 的。

7、这个工具是程序员写给程序员的工具，因此，很多设计都是沿着编程习惯来，大家可以从重载、函数命名、双数据区设计来仔细体会这些特点。

笔者这里补充这些内容，主要的目的就是帮助大家拓展思维，写程序很多时候不要为写而写，不要为了规则写程序，很多规则，哪怕本书讲解的 C 和 C++ 无错化程序设计方法，都是为了更好得完成工程开发服务的，规则本身不是目的。

因此，有个原则请大家注意：**所有的规则都是为需求服务的，当需求与规则冲突，打破规则，满足需求。**

再补充一点，笔者一直有个看法，**程序员也是用户**。程序员是自己代码的用户，后开发的代码是以前代码的用户，每次代码的开发，既要考虑最终用户的需求，也要考虑后来使用这段代码的程序员的需求，开发简便，程序才不容易出 bug。

但程序员又不是严格意义上的用户，因为程序员比用户多一点能力，修订代码，将代码与开发需求做适应性调整的能力，因此，笔者强调，如果一段代码，api 开发得不是很好用，很费解，好的程序员，不是去死记硬背 api，严格按照 api 办事，因为死记硬背的东西，也可能会搞忘，或者记忆模糊，在将来的某一天，就可能出 bug。

笔者的习惯是先学习，通过阅读源代码和大量的做实验，仔细摸清楚 api 的每个特性，然后，按照自己的习惯，重新做一层封装，根据自己的开发需求，有意识隐蔽一些无关的细节，开发一些需要的细节，最终使 api 变得更加好用，更加不容易出错，这是最关键的。

举个例子，当年笔者使用 ODBC 接口开发 access 的数据库程序，感觉 ODBC 就不是很好用，里面特别有一个例子，如果一个新表中没有任何数据，第一次添加数据会出现异常。

笔者于是用了将近半年的时间，将 ODBC 做了自己的二次封装，之后笔者的程序都使用这个二次封装接口，十分方便，也没有再出现过 bug。这实际上是笔者自己为自己开发了一个数据库 api。

这看起来有点得不偿失，因为当时的项目开发周期都没有半年，笔者完全是使用业余时间完成的这个工作。但随即，这种设计带来了另外一个好处。

笔者后续的一个开发任务，ReportMaster，一个报表管理系统，需要在 20 亿条数据中检索数据并整理成报表打印，为了提升效率，笔者用了很多底层优化，这中间就使用了磁盘和内存两级缓存，以哈希表方式的快速检索遍历等，由于使用数据库本身不方便，笔者被迫是 C 直接访问磁盘文件的方式，另行构建了一套高速数据存取系统。

但由于有上述笔者自己的 api，笔者的报表业务代码在从 ODBC 访问，改为普通的文件访问时，没有做任何更动，所有的修改都是在 api 下层做的，上层业务代码感觉不到这种修改，仅仅是速度增加了。这也是 api 的威力。

各位读者在这个过程中，能体会到什么吗？

9.5.3 任务池运行体的设计原理

笔者设计任务池运行体，原意就是简化任务池回调任务的开发，回避掉复杂的 switch 结构，但随着开发的进展，笔者越来越发现，这个运行体虽然定义为一个工具类，但其含义决不仅仅是一个简单的方便开发，其体现了更高一级的任务抽象观点。

前文我们详细论述了线程、任务的基本形态，归根到底，有两种，一种是单流程任务，即完全顺序执行，执行到最后，函数退出，任务完成。另一种是前面几次讨论的守候循环结构，即“初始化----循环体----结束”这种结构。

然在任务池运行体的设计中，笔者认为这两种还可以进一步抽象。笔者认为，**任务就是描述业务逻辑的代码片段，根据业务，这段代码可以在一个时间片中完成，也可以在多个时间片中完成，多任务运行环境应该同时支持这两种开发需求。**

但另一方面，每个任务片段里面，可能有循环，也可能没有循环，多任务运行环境没有必要，也不能去规定任务代码，有，或者没有循环结构，这些控制应该交由任务代码自己来决定。

换言之，笔者准备彻底打破从线程池就开始的一个推论，即业务流程中包含循环。而准备以一套更加灵活的机制，让线程的开发者，自己来决定，是不是有循环，以及循环是不是继续。

这也就是说，任务池的执行机制，将不再对任务的开发者，做任何代码规范上的建议和限制。这将给任务开发提供更大程度的灵活性。

9.5.3.1 任务池运行体的核心设计思想

在前文任务池的介绍中，笔者给大家展示了一段标准任务模型，应该说，这段任务模型还是比较复杂的。

```
static bool TaskCallback(void* pCallParam,int& nStatus)
{
    //...
    switch(nStatus)
    {
        case 0:      //init      //初始化代码
            //...
            nStatus++;
            break;
        //...
    }
    return true;      //正常情况下，返回真，任务继续
}
```

任务池运行体的核心设计思想，就是取消掉任务函数中庞大的 **switch** 语句，而将每个 **case** 分支，看做一个可以拆分的任务时间片需求，即一个程序片段，允许客户单独写出一个调用函数。

同时，该调用函数也允许程序员通过返回值的 **true** 或 **false**，控制循环的继续或跳出，这比较符合常规的 C 程序书写习惯。

9.5.3.2 任务池运行体的一对多模型

在构建任务池运行体模块时，笔者认为有必要分清楚两个概念：任务和任务片段。我

图 9.22：原始的任务池模型

们用下面的图示来说明。

在原始的任务池中，管理的最小单元是任务，至于每个任务里面是如何构成的，任务池有建议，但是没有管理。而在任务池运行体中，我们的管理粒度明显细化了，我们把任务进一步细化为任务片段，并且，任务池运行体理解这种任务片段的含义，能和这些片段代码互动运行。如下图所示：

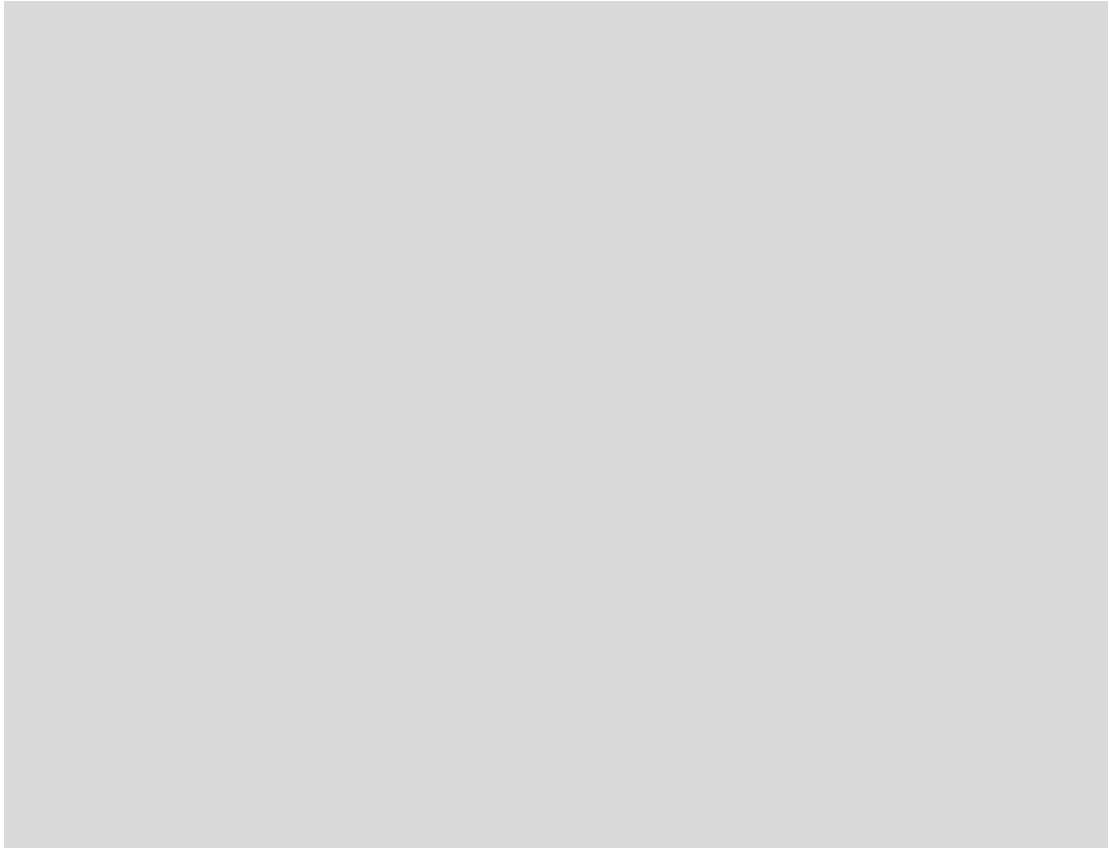


图 9.23：细化了任务片段的任务模型

因此，这里面的任务池运行体有个一对多的关系，这有两层意思：

1、针对任务池而言，一个任务池，能同时并发运行多个任务，任务池运行体同样也具有这个特性。

2、针对一个任务而言，任务池认为其是唯一的整体，而任务池运行体认为其是一系列顺序排列的任务片段的组合。

9.5.3.3 结束----一个歧义的产生

正是因为就任务池运行体而言，任务被细化为一系列片段的组合，这就带来一个很具体的问题，如何终止一个任务。

在任务池的开发设计中，这个问题比较简单，因为每个任务就是一个函数，不断被重入调用。那么，当系统结束时，终止任务就很容易，停止调用就可以了。

但在任务池执行体中，我们知道，任务片段通常还是遵循三段式，即“初始化----循环体----结束”，并且，程序员通常的开发习惯，会在初始化代码动态申请资源，开启二元动作的第一步，在结束代码中释放资源，完成二元动作的第二步。

这就带来一个问题，任务池运行体中的任务，应该怎么结束才能使任务逻辑完整，不至于泄漏资源。

经过思考，笔者认为**任务池运行体的一个任务，必须按照顺序，将其每个片段都至少执行一次，这个任务才算是执行完毕**。任务池运行体的程序开发，必须体现这一特性，不能简单粗暴退出。

9.5.4 任务池运行体的类声明

经过多方考虑和规划，笔者最终设计出较为理想的任务池运行体模块。其类声明如下：

```
class CTonyBaseLibrary;
class CTonyTaskRun
{
public:
    //构造函数很简单，就是保存聚合工具类指针备用即可。
    CTonyTaskRun(CTonyBaseLibrary* pTonyBaseLib)
    {m_pTonyBaseLib=pTonyBaseLib;}
    //析构函数也是简单关闭所有任务。
    ~CTonyTaskRun() {StopAll();}
public:
    //启动一个任务，这里有多重重载，方便调用
    bool StartTask(
        _TASKPOOL_CALLBACK pCallback, //任务片段回调函数
        void* pCallParam=null,        //回调函数参数指针
        char* szAppName=null);        //应用名（可以为空）
    //利用 Info 描述启动多次任务
    bool StartTask(
        STonyTaskRunInfo* pInfoStruct, //任务描述核心数据结构体
        char* szAppName=null);        //应用名（可以为空）
    bool StartTask(
        CTonyTaskRunInfo* pInfoObject, //任务描述对象指针
        char* szAppName=null);        //应用名（可以为空）
    //停止所有任务，退出时用，注意，这里是温和退出，每个任务的每个片段都将得到执行
    void StopAll(void);
    //工具函数，判断是否在运行中
    bool IsRunning(void) {return m_ThreadManager.ThreadContinue();}
    //工具函数，获得线程总数计数
    int GetThreadCount(void) {return m_ThreadManager.GetThreadCount();}
    //工具函数，打印内部信息，协助 debug 或性能观测
    void PrintInfo(void);
private:
    //最核心的任务执行回调函数，这实际上是任务池的一个任务回调
    //但其内部逻辑实现了代码片段到完整任务的转换
    static bool TonyTaskRunTaskCallback(void* pCallParam,int& nStatus);
    //这里使用了线程控制锁简化操作
    CThreadManager m_ThreadManager;
    //保存的聚合工具类指针
    CTonyBaseLibrary* m_pTonyBaseLib;
};
```

9.5.5 StartTask

StartTask 是任务池运行体很重要的功能函数，它有一个隐含约定，当第一次被调用，任务池运行体就视为启动，直到 **StopAll** 被调用，才结束。

但处理范围上，它每次只启动一个任务，即一个片段组，这个函数本身可以多次重入被调用，以便启动多个任务。

这个函数有三个重载，方便使用。但注意，任务的启动不是一定成功的，比如任务池任务总数满了，新的任务就无法注册，因此，这个函数返回 `bool` 量，成功返回真，否则返回假。调用者有责任检查这个函数的调用值。

同任务池一样，本类也有自己的线程任务管理变量，这是通过线程管控锁完成的，目的就是要把本对象所属任务与任务池的任务隔离开，本对象可以单独 `StopAll`，而不会影响任务池其他任务。

请注意，本类使用的回调函数构型，和任务池一模一样，这种模糊，就是为了实现透明设计，应用层设计一个任务回调函数，既可以使用本类对象管理启动，也可以直接调用任务池，实现最大的方便性。

```
//本重载主要应对那些单任务片段的任务，即只有一个回调函数的任务，可以直接注入启动。
bool CTonyTaskRun::StartTask(_TASKPOOL_CALLBACK pCallback,
                             void* pCallParam,
                             char* szAppName)
{
    //先实例化一个任务描述对象，调用下面的函数完成注册任务
    CTonyTaskRunInfo InfoObj;
    InfoObj.AddTask(pCallback,pCallParam);    //直接将任务数据添加到任务描述
    return StartTask(&InfoObj,szAppName);
}

//本重载主要应对使用任务描述对象，并完成了任务描述的应用，注入启动任务
bool CTonyTaskRun::StartTask(CTonyTaskRunInfo* pInfoObject,
                             char* szAppName)
{
    return StartTask(pInfoObject->GetInfoPoint(),szAppName);
}
```

```

//本函数利用任务描述对象内部的数据结构体，完成最终的任务启动工作
//请注意，本函数使用了远堆传参，来启动任务。远堆传参的参数，就是前文所述的结构体
//STonyTeskRunTaskCallbackParam
bool CTonyTaskRun::StartTask(STonyTaskRunInfo* pInfoStruct,
                             char* szAppName)
{
    bool bRet=false;                                //准备返回值
    if(!m_ThreadManager.ThreadContinue())           //如果本对象未启动，则启动
        m_ThreadManager.Open();
    //请注意这里，开始准备远堆传参的参数结构体数据区，使用了内存池动态申请。
    //内存池的指针是从聚合工具类中获得的。
    STonyTeskRunTaskCallbackParam* pParam=(STonyTeskRunTaskCallbackParam*)
        m_pTonyBaseLib->m_pMemPool->Malloc(
            STonyTeskRunTaskCallbackParamSize,
            "CTonyTaskRun::pParam");
    if(pParam) //内存申请成功，继续执行，否则返回假
    {
        pParam->m_pTonyBaseLib=m_pTonyBaseLib;       //聚合工具类指针
        pParam->m_pThis=this;                         //本对象指针
        pParam->m_nRunIndex=0;                        //状态机归 0
        if(szAppName)                                //如果外部提供了应用名
            SafeStrcpy(pParam->m_szAppName,           //拷贝应用名
                szAppName,
                TONY_MEMORY_BLOCK_INFO_MAX_SIZE);
        else                                          //否则清空应用名
            TONY_CLEAN_CHAR_BUFFER(pParam->m_szAppName);
        //注意，这里使用了任务描述的粘合类特性，粘合到参数区的任务描述数据结构上
        CTonyTaskRunInfo InfoObj(&(pParam->m_Info));
        //目的是调用 CopyFrom 来拷贝外部传入的任务参数
        InfoObj.CopyFrom(pInfoStruct);
        //调用聚合工具类中的任务池对象，注册任务
        //注意，注册是本类提供的标准任务执行体，该任务会解析任务描述
        //提供二次执行，真正执行应用层的任务调用。
        bRet=m_pTonyBaseLib->m_pTaskPool->RegisterATask(
            TonyTeskRunTaskCallback,                 //这是本对象的任务
            pParam);                                  //应用层注册的任务在这里
                                                    //会被二次解析执行

        if(bRet)
        {
            m_ThreadManager.AddAThread();             //如果成功，线程计数器+1
            if(szAppName)                             //如果有应用名，打印提示
                XGSyslog("%s Start...\n",pParam->m_szAppName);
        }
    }
    return bRet;
}

```

9.5.6 StopAll 和 PrintInfo

StopAll 相对就比较简单了，直接利用线程控制锁完成逻辑。

```
void CTonyTaskRun::StopAll(void)
{
    m_ThreadManager.CloseAll();
}
```

PrintInfo 也比较简单，这里一并介绍，就是打印一些关键参量。

```
void CTonyTaskRun::PrintInfo(void)
{
    TONY_XIAO_PRINTF("task run: task count=%d\n",
        m_ThreadManager.GetThreadCount()); //打印线程总数
}
```

9.5.7 任务执行线程回调

任务线程执行回调，是本类最重要的功能，它实际上是一个解析器，把任务描述结构描述的任务片段序列，解析成任务池需要的一个完整任务函数，最终完成功能。

请注意，本函数是任务池定义的标准回调函数构型。也就是说，这是标准的任务。另外，由于任务池的任务是多次重入的，本函数也是多次重入的。因此，本函数需要长期保存的变量，均放在参数结构体中，参数结构体的生命和本函数等长，只有任务结束，最后退出时，才会释放参数结构体。

还有个重要细节请关注，任务池提供的状态机，nStatus，这里的含义变了，这是本函数的状态机，而不是业务层注册任务的状态机。这点非常“绕”，可能很多读者在这里都会感到费解。

我们这么来解释，本函数，就是任务池的一个任务，因此，本函数可以享用任务池的状态机服务。至于应用程序通过本类注册的任务，这是本类对外提供的服务，这个应用任务，是不理解任务池的概念的，它只知道本类的存在。

因此，任务池的状态机，对应用任务没有意义，应用任务的状态机，存在于参数结构体中，pParam->m_nRunIndex 这个变量，是其状态机。这是两层关系。

这好比操作系统提供了功能，我们用 C 语言在操作系统下，又写了 Basic 的解析执行程序，这时候，用户编写的 Basic 程序是不理解操作系统的，如果 Basic 程序需要操作磁盘文件，是不能直接调用操作系统的磁盘文件服务，只有我们的 C 解析程序，中间做二次解析，把 Basic 的磁盘文件请求，翻译成操作系统的磁盘访问服务，才能正确执行。如下图所示例子：



图 9.24: 拓扑比较图

通过上图大家可以看到，任务池运行体，实际上相当于一套解析器，将应用程序任务片段序列，翻译成任务池能识别的任务，并提供时间片执行。对于应用层而言，此时是看不到任务池的，只知道任务池运行体的特性。

因此，任务池的状态机服务，仅对任务池运行体的执行任务线程有效，应用程序任务片段序列的状态机，需要任务池运行体单独设计提供。`pParam->m_nRunIndex` 就是实现任务池 `nStatus` 相似功能的状态机服务。

```

bool CTonyTaskRun::TonyTaskRunTaskCallback(void* pCallParam,int& nStatus)
{
    bool bCallbackRet=false;           //记录用户任务片段的调用结果
    bool bGotoNextStatus=true;         //跳到下一状态的标志
    STonyTaskRunTaskCallbackParam* pParam= //强制指针类型转换，获得参数指针
        (STonyTaskRunTaskCallbackParam*)pCallParam;
    if(!pParam) return false;          //防御性设计，如果没有参数，终止
    CTonyTaskRun* pThis=pParam->m_pThis; //方便调用，获得本类指针 pThis
    switch (nStatus)                   //这个状态机是任务池提供的
    {
    case 0:                             ///这是本函数的主执行代码
        if(pParam->m_Info.m_nTaskCount > pParam->m_nRunIndex)
        { //只要应用层任务片段没有被轮询完毕，一直在本片段执行
            bGotoNextStatus=false;      //这个=false，就是不希望跳转
            //这里的回调请仔细看
            bCallbackRet=               //取得应用层片段的回调结果
                //根据应用层状态机 pParam->m_nRunIndex，调用合适的片段
                pParam->m_Info.m_CallbackArray[pParam->m_nRunIndex] (
                    pParam->m_Info.m_pCallParam, //这是透传的参数指针
                    pParam->m_nRunIndex);      //这里最关键，大家注意此处没有把
                                                //线程池状态机 nStatus 传入应用层
                                                //代码模块，而是使用自己的状态机
                                                //这种“欺骗”，是透明设计的关键

            //根据任务池的定义，如果应用程序代码片段返回 false，表示其希望跳到下一片段
            //因此，这里做应用程序状态机的+1 动作
            if(!bCallbackRet) pParam->m_nRunIndex++;
            //注意这个设计，这是非常关键的一步。当 StopAll 被调用，本对象即将退出时
            //本函数检测到这个结果，并不是简单把 nStatus+1 做退出
            //而是开始步进应用任务片段，即把应用程序的状态机 pParam->m_nRunIndex 累加
            //这是强制执行完应用任务片段序列的每一步。
            //最终，本任务会因为应用任务执行完毕而关闭，而不是在此因为系统退出关闭
            //这样保证不至于产生资源泄漏等情况。
            if(!pThis->m_ThreadManager.ThreadContinue())
                pParam->m_nRunIndex++;
        }
        //如果应用程序片段已经执行完毕，则本函数状态机+1，跳转到下一片段结束任务
        else bGotoNextStatus=true;
        break;
    default: //exit
        //如果有应用名，打印退出提示
        if(0<strlen(pParam->m_szAppName))
            pThis->XGSSyslog("%s Stop!\n",pParam->m_szAppName);
        pThis->m_ThreadManager.DecAThread(); //线程计数器-1
        pThis->m_pTonyBaseLib->m_pMemPool->Free(pParam); //释放参数结构体
        return false; //返回假，结束任务
    }
    if(bGotoNextStatus) nStatus++; //根据前文判断，调整本函数状态
    return true; //返回真，不退出任务
}

```

9.5.8 任务池运行体小结及调用示例

任务池运行体，可以说是 api 设计的经典之作。请注意，这里的 api，并不仅仅包含模块的调用接口等显式的方法函数构型一致，而是本类提供的任务回调服务，和任务池的回调服务，无论从函数构型，还是各种开发特性（如返回值定义，如状态机定义），都完全一模一样。

这表示，应用层开发一个任务回调函数，既可以把其作为一个单一任务，直接注册到任务池执行，也可以借用本类的定义，将其作为任务序列的某一个片段执行，都没有问题，应用程序从组织结构上，无需任何更改。这样保证了最大的调用灵活性。

当然，为了实现这一点，大家可以看到，笔者付出的代价也不小，在前文 `TonyTaskRunTaskCallback` 这个函数中，笔者做了大量的抽象思维定义，并使用一定的欺骗手段，在改变状态机的真实含义。

但所有这些工作都是值得的，在后文中，大家可以看到这种设计带来的巨大开发便利性。这里，笔者先给大家提示一点如何使用任务池运行体，实现任务开发的实例。

```
//由于实际任务开发较为复杂，依赖条件较多，我们这里假设一个聚合工具类指针
CTonyBaseLibrary* g_pTonyBaseLib;          //演示用的聚合工具类指针
class CTestTaskRun
{
private:
    //我们假设本对象的任务分三个片段，就是前文所述的三段式开发，代码此处略
    static bool TaskInit(void* pCallParam,int& nStatus);    //初始化
    static bool TaskLoop(void* pCallParam,int& nStatus);    //循环体
    static bool TaskExit(void* pCallParam,int& nStatus);    //结束代码
    CTonyTaskRun* m_pTonyTaskRun;                          //线程池运行体指针
public:
    CTestTaskRun()
    {
        //创建线程池运行体
        m_pTonyTaskRun=new CTonyTaskRun(g_pTonyBaseLib);
        //注意这里，设置一个描述对象
        CTonyTaskRunInfo tri;
        tri.AddTask(TaskInit,this);                        //请注意，依次添加，顺序不能乱了
        tri.AddTask(TaskLoop,this);
        tri.AddTask(TaskExit,this);
        m_pTonyTaskRun->StartTask(&tri);                    //启动任务
    }
    ~CTestTaskRun()
    {
        if(m_pTonyTaskRun)
        {
            m_pTonyTaskRun->StopAll();                      //结束所有任务
            delete m_pTonyTaskRun;                          //释放资源
            m_pTonyTaskRun=null;
        }
    }
};
```

9.6 时间片小结

时间片划分是现代多任务操作系统的理论核心，也是并行计算的理论支撑。深刻理解时间片的含义，特性，对于程序员开发多任务应用程序，是至关重要的一步。

大家可能注意到，笔者在本章中，并没有就事论事，讨论线程具体怎么操作，线程函数怎么书写等细节。而是花费了大量的时间和精力，和大家一起讨论时间片的由来，多任务操作系统的实现机制。

这是有原因的，在很多讲述语言的教科书上，很少涉及到多任务开发，而讲述操作系统特性的教科书，如《Windows 程序设计》，又主要针对操作系统的特性做讲解，长期以来，笔者发现，几乎找不到一本，专门论述多任务并行开发的书籍，来帮助程序员深刻理解并行编程与单任务编程的异同。这已经成为几乎所有程序员的一块“短板”。

而另外一方面，笔者长期的工程实践经验表明，并行开发难度非常大，其中要考虑争用、碰撞等数据安全访问机制，要考虑时间片，要考虑任务协调性，动作规避等等一系列问题。

这要求程序员不仅仅拥有熟练的语言使用技巧，还需要对操作系统，数据结构，算法，甚至编译原理都有很深入的了解，能深刻理解时间片的由来以及特性，并能将这些知识应用到工程设计实战中去，才可能做好并行开发。

因此，笔者在写作本书时，一开始就决定，需要针对并行开发做深入的讨论和研究，将笔者这么多年研究的一点心得分享给大家，帮助尽可能多的程序员，迅速理解并行计算开发，理解时间片，理解多任务操作系统提供的运行环境，最终帮助大家实现并行工程开发。

这里仍然要提醒大家一点，虽然本章、本书会花大量的篇幅，讨论并行计算，但笔者认为，并行开发最困难的，不是学习，而是思维习惯。

这类开发通常要求程序员从工程设计一开始，就拥有系统设计思维，针对任务的协调性要做前期的计算和分析，这甚至要求精确到每个时间片去评估其效率，逻辑正确性，这种能力是需要经过长期的训练和实战才可能产生的。

因此，建议各位读者在阅读本书的基础上，应尽量多学习一些开发的基础知识，并不断实战演练，方能开发出强健、安全、合用的并行工程程序。

最后，笔者通过本文展示的代码，还希望各位读者能深刻理解程序设计的一些开发思维：

1、api 的构成，api 是一个逻辑概念，就是应用程序的业务层和功能层之间的接口约定，但请大家注意，这个接口约定不仅仅包含显式的接口函数定义，如本章所示，还包括很多隐式的特性，如任务回调函数的返回值约定，状态机 nStatus 的特性定义等，希望各位读者深刻理解这一点，在以后的工程设计中关注这些细节。

2、api 的意义。笔者理解 api，就是“一刀切”。api 的存在，割裂了应用程序业务层和功能层之间的耦合联系，这使得两个模块可以分别开发，升级维护，保证了开发的灵活度。另外，如本章所示，我们甚至可以在遵循 api 的前提下，在原有的两层之间，增加第三层，甚至第四层中间层，这样逐级细化，可以在不改动原有代码的基础上，增加很多功能和特性，这种开发灵活性，必须靠 api 来保证。

3、CTonyTaskRun 实际上就是在任务池的基础上，二次增加了一层解析层，允许应用程序把任务切割成独立的程序片段，CTonyTaskRun 会把这些片段重新解析为一个任务池任务加以执行。从某种意义上讲，CTonyTaskRun 甚至类似于某些解析执行的语言，自己定义了一套语法，允许应用层以新的语法运行，CTonyTaskRunInfo 就是这种二进制语法的集中体现，因此，请各位读者仔细体会其中用到的编译原理的理念。

4、抽象，还是抽象。笔者理解，所谓抽象，就是不断提取事物的共性，使用同一段逻辑流程来处理同一共性，这也是程序设计的核心理念，本章可以说通篇的核心主题，就是抽象。请各位读者仔细体会这种抽象性思维，以及工程开发中的抽象习惯，不断优化自己的代码。

5、割裂，本章说了很多割裂，时间的割裂，产生了时间片，程序的割裂，产生线程，线程的割裂，产生任务，任务的割裂，产生任务片段，上层应用逻辑、业务逻辑与下层功能层的割裂，产生 api，建议各位读者仔细体会这种“割裂”思维，很多时候，这是抽象的关键。

6、逻辑的完整性，执行的片段性，这可以说是多任务运行环境下，每个并行任务的关键特征，建议各位读者要仔细理解这个特征，并代入程序设计思维中，在这个思想的指导下完成设计。很多时候，程序员开发不好并行程序，主要就是因为没有关注到这个片段性。

- 第 10 章 Log 日志管理系统
 - 10.1 日志管理系统需求分析
 - 10.2 设计原理和边界定义
 - 10.3 类声明
 - 10.4 构造函数和析构函数
 - 10.4.1 构造函数
 - 10.4.2 析构函数
 - 10.5 文件名控制逻辑
 - 10.6 业务输出方法函数
 - 10.7 Log 日志系统小结

第 10 章 Log 日志管理系统

在前文我们讲过，log 日志是商用数据传输系统的必备功能，不管是开发时的 Debug，还是性能调试，以及运行时状态监控，都是非常重要的观测手段。在前文，笔者也为大家展示了一套底层的 Debug 功能模块 CLowDebug，这也是笔者较常用的一套建议日志跟踪系统。但在后来的一次测试中，笔者发现了问题。

笔者的一个服务器程序，开发完成后运行，一切 OK，但是，几个小时后，挂死了，所有的线程挂死，服务停止。笔者杀死进程后重启，情况又恢复正常，但几个小时后，又出现了挂死。

这是个严重问题，笔者开始跟踪，但是，无论查找哪段代码，Code Review 都没有问题，笔者百思不得其解。知道笔者开始翻看工程运行目录下的磁盘文件，发现了问题，dbg 文件变得很大，差不多 2G 左右。当时笔者还没有意识到这是问题点，只是直觉上，这么大的文件，可能会有点问题。

然后笔者检查了程序代码，发现最近的几次 debug 行为，为了观测 bug 点，笔者增加了很多高频的 debug 输出，信息量很大，然后查了资料，发现 32 位 Linux 下，一个文件最大一般也就 2G，超过这个限额，所有的文件写入要求被系统悬挂，无法执行。

这就找到了问题，笔者本身的代码，并没有问题，但问题是，日志文件太长了，超过了系统的限制，系统为了保护磁盘访问模块不至于崩溃，强行阻塞所有的写操作，而笔者的程序，几乎每段函数都有日志输出，导致挂死。

这样，笔者发现有必要在 Debug 功能模块之上，单独开发一套专业的 log 日志系统。

值得一提的是，在笔者决定开发这套日志系统时，有朋友建议采用开源的 log4c++ 系统直接替代，笔者做了测试，但发现这套开源系统虽然也功能很强大，也能跨平台兼容，但是有一个问题，就是太大了，单独的一个静态链接库，就达到 1.8M，而笔者很多服务器程序本身，编译后的可执行代码都不超过 300k，这显然难以接受，并且，在嵌入式系统连入这么大的静态库，也太奢侈了一点。因此，笔者最终决定，还是自行开发。

10.1 日志管理系统需求分析

首先，笔者根据自己的服务器开发经验，整理了一下日志系统的需求：

- 1、提供类似 Debug 工具的日志输出功能，要求可以同时输出到控制台和文件，并可选。
- 2、每行日志输出，自动添加时间戳，方便后期观察分析。
- 3、初步决定分为四级日志：
 - a) Syslog，常规的事件记录，所有的错误信息
 - b) Debug，常规的调试信息，信令跟踪输出（可选）
 - c) Debug2，较高频度的业务事件，如 Socket 连接事件，失败事件说明等
 - d) Debug3，最高频度的业务事件，如内存池的申请和释放，线程池线程的起停等。
- 4、支持多文件日志系统，根据时间和大小自动切割日志文件大小，形成连续的文件序列，而不是单个的大型文件，防止文件体积超限，导致挂死。
- 5、文件名要有时间戳，方便程序员和网管人员查找信息。
- 6、原则上 1 小时产生一个新的日志文件，这样信息量基本不会超过 2G 的限额。
- 7、有文件总数限制管理，原则上默认保留最后 72 小时（可修改）的日志内容，超时部分自动删除。以保证磁盘总量不超限。
- 8、日志系统本身线程安全，方便多任务并行调用。

10.2 设计原理和边界定义

日志系统看似简单，但笔者真的将其作为一个工程项目设计起来，发现问题还是蛮多的。

首先，多文件日志本身实现不难，笔者以时间戳的小时级精度作为文件名的生成器，那么，自动的，每个小时就是一个文件。17:xx 的信息和 18:xx 的信息，自然就写不到一个文件中。

不过这里面有个要求，保留最后 72 小时的文件，这要求文件名本身在程序内部需要以一个列表来保存，以便做筛选和删除。也就是说，文件名不能失控，必须随时被模块管理着。笔者考虑了一下，准备在内部做一个小队列数组来解决这个问题。

但这又带来另外一个问题，服务器程序可能会中途退出维护，上一次运行和下一次运行，这个日志文件信息不能丢失，因此，上述队列数组本身，也需要一个文件来做长期保存，不能简单放在内存里面。

其次，笔者发现就算以 1 小时为界，但在某些需要 debug 的特殊环节，仍然需要打印很多高频信息，很可能在 1 小时中，已经信息量已经超过了 2G 的限制，导致挂死。因此，除了 1 小时切换一次文件名的机制外，还需要有个文件大小上限，超过一定尺寸后，一定要换新文件名，否则就很危险。这个大小，暂定 2G 的一半，1G。

这说明，一个日志文件，有几个终结点，1 小时满了，换文件名，超过 1G，换文件名，另外，每次关闭系统，重新再启动，也会换一个文件名。

还有一个边界也要考虑，即每行信息的大小，由于在数据库中，很可能会将 SQL 命令作为日志打印跟踪，笔者暂定以 2k 作为每行的上限，必要时可以调整大小。

另外，还要考虑，除了文本方式输出外，还应该设置二进制方式输出，否则在跟踪某些二进制信令方面，很不方便。

以上就是笔者在设计 log 日志系统时的设计考虑，因此，笔者首先作了基本的数据边界定义。

```
#define LOG_FILE_SIZE_MAX (1*1024*1024*1024) //每个日志文件最大 1G
#define LOG_ITEM_LENGTH_MAX (2*1024) //单条 log 最大长度 2k
#define LOG_FILE_CHANGE_NAME_PRE_SECONDS (60*60) //日志文件 1 小时更换一次名字
#define LOG_FILE_INFO_BUFFER_SIZE (256*1024) //日志目录最大长度(超长删除)
#define LOF_FILE_DEFAULT_HOLD 72 //一般保留 3 天的数据
#define LOG_TIME_STRING_MAX 128 //时间戳字符串长度
```

除此之外，笔者还不能不考虑到，并不是每个服务器都是 Linux 下的控制台输出，一个 Windows 的服务器如果使用这个模块，很可能并不是以 printf 方式打印到屏幕控制台，而很可能需要拦截输出信息，输出到一个窗口中。

因此，笔者考虑了一下，觉得应该设置一个回调函数构型，必要时，应用程序可以通过这个回调函数拦截所有的日志输出。不过这个问题倒是比较好解决，因为在前文的 LowDebug 中已经有这个设计，笔者这里直接借用即可。

```
typedef void (*_APP_INFO_OUT_CALLBACK)(char* szInfo,void* pCallParam);
```

10.3 类声明

log 系统的类声明还是比较简单的，因为其作为使用频率最高的日志系统，使用方便性是最基本的要求，仅提供有限的公开方法函数。

```
//笔者的日志管理系统，由于很多系统都有自己的日志系统，此处以笔者的英文名修饰，避免重名
#define FILENAME_STRING_LENGTH 256 //文件名长度
class CTonyXiaoLog
{
public: //静态工具函数类
    //定制时间戳字符串
    static int MakeATimeString(char* szBuffer,int nBufferSize);
public: //构造函数和析构函数
    CTonyXiaoLog(CTonyLowDebug* pDebug, //debug 对象指针(log 也需要 debug)
        CTonyMemoryPoolWithLock* pMemPool, //内存池指针，内存队列要用
        char* szLogPath, //日志路径
        char* szAppName, //应用名(修饰日志文件)
        int nHoldFileMax=LOF_FILE_DEFAULT_HOLD, //保留多少个文件
        bool bSyslogFlag=true, //日志级别开关，true 打开，否则关闭
        bool bDebugFlag=true,
        bool bDebug2Flag=false,
        bool bDebug3Flag=false,
        bool bPrintf2ScrFlag=true); //是否打印到屏幕开关
    ~CTonyXiaoLog(); //析构函数

public: //公有输出方法
    void _XGDebug4Bin(char* pBuffer,int nLength); //二进制输出
    int _XGSyslog(char *szFormat, ...); //Syslog 输出，变参函数
    int _XGDebug(char *szFormat, ...); //Debug 输出，变参函数
    int _XGDebug2(char *szFormat, ...); //Debug2 输出，变参函数
    int _XGDebug3(char *szFormat, ...); //Debug3 输出，变参函数
```

```

public:    //开关变量，对应构造函数的开关
    bool m_bSyslogFlag;
    bool m_bDebugFlag ;
    bool m_bDebug2Flag;
    bool m_bDebug3Flag;

private:  //内部功能函数
    int _Printf(char *szFormat, ...);           //最核心的打印输出模块，变参函数
    void DeleteFirstFile(void);                 //删除最老的文件
    void FixFileInfo(void);                     //修订文件名目录队列
    void MakeFileName(void);                    //根据时间和文件大小，定制文件名
    void GetFileName(void);                     //获得当前可用的文件名

private:  //内部私有变量区
    CMutexLock m_Lock;                          //线程安全锁
    char m_szFilePath[FILENAME_STRING_LENGTH]; //文件路径
    char m_szFileName[(FILENAME_STRING_LENGTH*2)]; //文件名
    unsigned long m_nFileSize;                  //当前文件大小
    time_t m_tFileNameMake;                    //定制文件名的时间戳
    int m_nHoldFileMax;                         //保留文件的数量，构造函数传入
    _APP_INFO_OUT_CALLBACK m_pInfoOutCallback; //应用程序拦截输出回调函数
    void* m_pInfoOutCallbackParam;              //透传的回调函数指针
    bool m_bPrintf2ScrFlag;                     //是否打印到屏幕的开关
    char m_szFileInfoName[FILENAME_STRING_LENGTH]; //文件名
    CTonyLowDebug* m_pDebug;                    //Debug 对象指针
    CTonyMemoryPoolWithLock* m_pMemPool;        //内存池对象指针
    CTonyXiaoMemoryQueue* m_pFileInfoQueue;     //文件名队列
};

```

10.4 构造函数和析构函数

日志系统的构造函数和析构函数主要还是完成系统的初始化和结束代码功能，但是，此处有个很重要的细节需要关注，日志文件的目录，需要在关机时写入磁盘，在开机是读入内存，必须在此完成。

我们前文介绍队列时已经说明，MemQueue 内部就有磁盘 Dump 的功能，因此，此处可以直接调用该功能函数完成。

另外，笔者在这里还有个推论，即一个应用程序如果需要拦截输出，这个程序必然也拦截了 LowDebug 的输出，因此，拦截回调函数，笔者不准备在构造函数的参数列表中输入，而是直接从 Debug 对象中读取。

10.4.1 构造函数

构造函数完成所有的初始化工作。

```

CTonyXiaoLog::CTonyXiaoLog(CTonyLowDebug* pDebug,          //参数介绍略
                           CTonyMemoryPoolWithLock* pMemPool,
                           char* szLogPath,
                           char* szAppName,
                           int nHoldFileMax,
                           bool bSyslogFlag,
                           bool bDebugFlag,
                           bool bDebug2Flag,
                           bool bDebug3Flag,
                           bool bPrintf2ScrFlag)
{
    m_pDebug=pDebug;          //保留 Debug 对象指针
    m_pMemPool=pMemPool;      //保留内存池指针
    //请注意，这里从 Debug 对象中获得拦截回调函数信息
    m_pInfoOutCallback=m_pDebug->m_pInfoOutCallback;
    m_pInfoOutCallbackParam=m_pDebug->m_pInfoOutCallbackParam;
    //利用 debug 输出启动标志
    TONY_DEBUG("CTonyXiaoLog: Start!\n");
    //获得日志文件名基准字符串，这里主要使用输入的路径名和应用名生成基本名
    //如路径是“/var”，应用名是“test_project”，
    //则基准名为“/var/test_project”，
    //这样，以后的文件名，就是在这个基本名后加时间戳实现
    //如：/var/test_project_Thu_Jul_16_14_31_44_2009.log
    FULL_NAME(szLogPath,szAppName,m_szFilePath,"");
    //获得日志文件名目录的保存文件名
    //如：/var/test_project.info
    FULL_NAME(szLogPath,szAppName,m_szFileInfoName,"info");
    //清空当前文件名缓冲区
    TONY_CLEAN_CHAR_BUFFER(m_szFileName);
    //当前文件尺寸设置为 0
    m_nFileSize=0;
    m_bSyslogFlag=bSyslogFlag;          //保存 Debug 级别开关变量
    m_bDebugFlag =bDebugFlag ;          //为 false 的级别将不会被输出
    m_bDebug2Flag=bDebug2Flag;
    m_bDebug3Flag=bDebug3Flag;
    m_bPrintf2ScrFlag=bPrintf2ScrFlag;  //保存屏幕输出开关
    m_nHoldFileMax=nHoldFileMax;        //保存最大保留文件个数
    m_pFileInfoQueue=new CTonyXiaoMemoryQueue( //实例化文件目录队列对象
        pDebug,
        m_pMemPool,
        "CTonyXiaoLog::m_pFileInfoQueue");
    if(m_pFileInfoQueue)                //如果创建成功，注册到内存池
    {
        m_pMemPool->Register(m_pFileInfoQueue,
            "CTonyXiaoLog::m_pFileInfoQueue");
    }
    m_pFileInfoQueue->ReadFromFile(m_szFileInfoName); //读入上次保留的文件名信息
    MakeFileName();                          //根据当前时间戳，定制一个文件名
}

```

这里需要特别给大家介绍一下 FULL_NAME 这个函数型宏。

```

#ifdef WIN32
    #define PATH_CHAR "\\\"      //Windows 下使用'\'作为路径间隔符
#else // not WIN32
    #define PATH_CHAR "/"      //Unix, Linux 下使用'/'
#endif
//参数表:
//path: 路径
//name: 主文件名 (file.ext 中的 file)
//ext_name: 副文件名 (file.ext 中的 ext)
//fullname: 最后生成的全文件名缓冲区指针
#define FULL_NAME(path,name,fullname,ext_name) \
{ \
    if(strlen(path)) \
    { \
        if(strlen(ext_name)) \
            SafePrintf(fullname,FILENAME_STRING_LENGTH, \
                "%s%s%s.%s",path,PATH_CHAR,name,ext_name); \
        else \
            SafePrintf(fullname,FILENAME_STRING_LENGTH, \
                "%s%s%s",path,PATH_CHAR,name); \
    } \
    else \
    { \
        if(strlen(ext_name)) \
            SafePrintf(fullname,FILENAME_STRING_LENGTH, \
                "%s.%s",name,ext_name); \
        else \
            SafePrintf(fullname,FILENAME_STRING_LENGTH,"%s",name); \
    } \
}

```

虽然看起来比较复杂,但其逻辑还是比较简单,就是调用 **SafePrintf** 函数构建文件名,其构建的结果见构造函数中的例子。

10.4.2 析构函数

析构函数主要做结束收尾工作,摧毁动态创建的队列对象,回收资源,把文件目录信息保留到磁盘上等。

```

CTonyXiaoLog::~CTonyXiaoLog()
{
    if(m_pFileInfoQueue)          //清楚文件目录队列对象
    {
        m_pFileInfoQueue->Write2File(m_szFileInfoName);    //清除前先保留到磁盘
        m_pMemPool->UnRegister(m_pFileInfoQueue);    //反注册对象指针
        delete m_pFileInfoQueue;          //删除对象
        m_pFileInfoQueue=null;
    }
    TONY_DEBUG("CTonyXiaoLog: Stop!\n");    //Debug 输出
}

```

10.5 文件名控制逻辑

文件名控制逻辑是 Log 日志系统中相对比较复杂的一块，我们先理一理它的逻辑关系。

1、每次打印一行信息到文件之前，我们需要先用 `GetFileName` 获得当前的文件名，然后打开文件，写入信息，关闭文件。

2、但 `GetFileName` 的时候，需要做两个检查，一个是当前时间与文件名产生的时间超过 3600 秒，即 1 小时，就调用 `MakeFileName`，以当前时间戳创造个新文件名。另一个是如果文件大小超过 1G，也调用 `MakeFileName` 创建新文件名，这样保证了文件大小不超限。

3、在 `MakeFileName` 中，我们会调用一次 `FixFileInfo`，检查文件名队列元素是不是超过 72 个，如果超过，就弹出第一个文件名，删除，以此保证 Log 文件的总个数不超标。

4、如果 `FixFileInfo` 发现文件名个数超标，则调用 `DeleteFirstFile` 删除第一个旧文件。实现队列的“推出”。

这里面有个小小的技巧，我们知道，维护 Log 文件总个数不超标，这算一个事件了，传统的做法，一般是做个守候线程，不断检查，但在这里这么做，显然太浪费了，没有必要。

因此，笔者做了一个推论，每次增加一个新文件名，才可能导致所有的文件名超标，因此，在 `MakeFileName` 这个点做次联动，实现超标删除即可，平时由于文件名没有增加，没有必要去检查做无用功。

10.5.1 GetFileName

```
void CTonyXiaoLog::GetFileName(void)    //获取当前文件名
{
    time_t tNow;                        //当前时间戳变量
    unsigned long ulDeltaT=0;           //Δt 变量
    if('\0'==m_szFileName[0])           //如果是第一次启动，文件名为空
    {
        MakeFileName();                 //无条件创造个文件名，返回
        goto CTonyXiaoLog_GetFileName_End_Porcess;
    }
    time(&tNow);                         //求得当前时间
    ulDeltaT=(unsigned long)tNow-m_tFileNameMake; //计算Δt
    if(LOG_FILE_CHANGE_NAME_PRE_SECONDS<=ulDeltaT)
    {
        MakeFileName();                 //如果Δt 超过 3600 秒，创造文件名返回
        goto CTonyXiaoLog_GetFileName_End_Porcess;
    }
    if(LOG_FILE_SIZE_MAX<=m_nFileSize) //如果当前文件大小尺寸超出 1G
    {
        MakeFileName();                 //创造文件名返回
        goto CTonyXiaoLog_GetFileName_End_Porcess;
    }
CTonyXiaoLog_GetFileName_End_Porcess:
    return;
}
```

10.5.2 MakeFileName

```

void CTonyXiaoLog::MakeFileName(void)           //创建一个新文件名
{
    char szTemp[LOG_ITEM_LENGTH_MAX];           //临时缓冲区
    MakeATimeString(szTemp,LOG_ITEM_LENGTH_MAX); //获得时间戳字符串
    FixFileInfo();                               //维护文件总个数不超标（默认 72 个）
    int nLen=SafePrintf(                         //注意看这句，利用构造函数中的种子名字
        m_szFileName,                           //加上时间戳，后面再加上“.log”后缀
        FILENAME_STRING_LENGTH*2,              //生成日志文件名
        "%s_%s.log",
        m_szFilePath,
        szTemp);
    nLen++;                                     //习惯，长度+1，保留最后'\0'的位置
    //将新的文件名添加到队列
    int nAddLastRet=m_pFileInfoQueue->AddLast(m_szFileName,nLen);
    if(0>=nAddLastRet)
    { //这是一个特殊的防护，如果队列满了（内存不够用），删除最开始三个文件名
      //释放内存空间，这是预防服务器业务太繁忙，导致内存不够用，队列无法添加的
      //规避措施，这也体现非关键模块为关键业务模块让路的思维
      DeleteFirstFile();
      DeleteFirstFile();
      DeleteFirstFile();
      //删除三个之后，重新尝试添加
      nAddLastRet=m_pFileInfoQueue->AddLast(m_szFileName,nLen);
      //如果此时添加仍然失败，投降，日志发生一点错乱没有关系。
    }
    m_nFileSize=0;                             //新文件创建，文件长度为 0
    //下面逻辑，新创建一个文件，在文件头先打印一点文件名相关信息，帮助以后的跟踪查找
    time(&m_tFileNameMake);
    { //由于这是非业务打印，因此不希望输出到屏幕，这里临时将屏幕开关关闭
      bool bPrint2Scr=m_bPrintf2ScrFlag;
      m_bPrintf2ScrFlag=false;
      _Printf("Tony.Xiao. base libeary log file %s\n",m_szFileName);
      _Printf("-----\n");
      m_bPrintf2ScrFlag=bPrint2Scr;             //输出完毕，屏幕开关恢复原值
    }
}

```

10.5.3 MakeATimeString

本函数是本类唯一一个静态纯工具函数，即使本类不实例化对象也可以执行。其逻辑非常简单，就是利用当前时间戳，以人可以阅读的格式，输出一个字符串，主要用于文件名的创建。


```

int CTonyXiaoLog::MakeATimeString(char* szBuffer,int nBufferSize)
{
    int i=0;
    time_t t;
    struct tm *pTM=NULL;
    int nLength=0;
    if(LOG_TIME_STRING_MAX>nBufferSize)           //防御性设计
        goto CTonyXiaoLog_MakeATimeString_End_Porcess;
    time(&t);                                       //获得当前时间
    pTM=localtime(&t);                             //或当当前时区的时间戳字符串
    nLength=SafePrintf(szBuffer,LOG_ITEM_LENGTH_MAX,"%s",asctime(pTM));
                                                    //时间戳字符串入缓冲区
    //localtime 生成的字符串最后自带一个'\n'，即回车符，这不利于后续的打印
    //因此下面这行向前退一格，清除掉这个回车符。这是一个小经验。
    szBuffer[nLength-1]='\0';
    //文件名有一定限制，一般不要有空格，':'字符，这里过滤一下，
    //将时间戳中的非法字符都改变成各系统都能接受的 '_' 下划线
    for(i=0;i<nLength;i++)
    {
        if(' '==szBuffer[i]) szBuffer[i]='_';
        if(':')==szBuffer[i]) szBuffer[i]='_';
    }
CTonyXiaoLog_MakeATimeString_End_Porcess:
    return nLength;                               //返回总长度
}

```

10.5.4 FixFileInfo

```

void CTonyXiaoLog::FixFileInfo(void)    //维护文件总个数不超标
{
    int nAddLastRet=0;
    //请注意，这里不是 if，而是一个 while，如果因某种原因，超标很多个文件
    //利用这个循环技巧，很轻松地将超标文件删除到只有 72 个，
    //很多时候，维护数组不超限，都是使用这个技巧
    while(m_pFileInfoQueue->GetTokenCount()>=m_nHoldFileMax)
    {
        DeleteFirstFile();
    }
}

```

10.5.5 DeleteFirstFile

```

void CTonyXiaoLog::DeleteFirstFile(void)    //删除第一个文件
{
    char szFirstFile[FILENAME_STRING_LENGTH]; //文件名缓冲区
    int nFirstFileNameLen=0;                //文件名字符串长度
    //从文件名数组中，弹出第一个文件名
    nFirstFileNameLen=m_pFileInfoQueue->GetAndDeleteFirst(
        szFirstFile,FILENAME_STRING_LENGTH);
    if(0>=nFirstFileNameLen)                //失败返回
        goto CTonyXiaoLog_DeleteFirstFile_End_Process;
    CTonyLowDebug::DeleteAFile(szFirstFile); //真实地删除文件
CTonyXiaoLog_DeleteFirstFile_End_Process:
    return;
}

```

10.6 业务输出方法函数

业务输出方法，就是包括前文所述的 Syslog 等方法，各应用模块调用这些方法，完成真实的日志输出功能。以下这几段函数都是线程安全的。

10.6.1 XGSyslog

```

int CTonyXiaoLog::_XGSyslog(char *szFormat, ...)
{
    //这段比较经典，变参函数处理模块，不再赘述
    char szBuf[LOG_ITEM_LENGTH_MAX];
    int nMaxLength=LOG_ITEM_LENGTH_MAX;
    int nListCount=0;
    va_list pArgList;
    va_start (pArgList,szFormat);
    nListCount+=Linux_Win_vsnprintf(szBuf+nListCount,
        nMaxLength-nListCount,szFormat,pArgList);
    va_end(pArgList);
    if(nListCount>(nMaxLength-1)) nListCount=nMaxLength-1;
    *(szBuf+nListCount)='\0';
    if(m_bSyslogFlag)                //如果开关打开
    {
        m_Lock.Lock();                //加锁
        {
            _Printf("%s",szBuf);        //真实执行打印
        }
        m_Lock.Unlock();                //解锁
    }
    return nListCount;                //返回长度
}

```

10.6.2 XGDebug

```

int CTonyXiaoLog::XGDebug(char *szFormat, ...)
{
    //这段比较经典，变参函数处理模块，不再赘述
    char szBuf[LOG_ITEM_LENGTH_MAX];
    int nMaxLength=LOG_ITEM_LENGTH_MAX;
    int nListCount=0;
    va_list pArgList;
    va_start (pArgList,szFormat);
    nListCount+=Linux_Win_vsnprintf(szBuf+nListCount,
        nMaxLength-nListCount,szFormat,pArgList);
    va_end(pArgList);
    if(nListCount>(nMaxLength-1)) nListCount=nMaxLength-1;
    *(szBuf+nListCount)='\0';

    if(m_bDebugFlag)                //如果开关打开
    {
        m_Lock.Lock();              //加锁
        {
            _Printf("%s",szBuf);    //真实执行打印;
        }
        m_Lock.Unlock();            //解锁
    }
    return nListCount;              //返回长度
}

```

10.6.3 XGDebug2

```

int CTonyXiaoLog::XGDebug2(char *szFormat, ...)
{
    //这段比较经典，变参函数处理模块，不再赘述
    char szBuf[LOG_ITEM_LENGTH_MAX];
    int nMaxLength=LOG_ITEM_LENGTH_MAX;
    int nListCount=0;
    va_list pArgList;
    va_start (pArgList,szFormat);
    nListCount+=Linux_Win_vsnprintf(szBuf+nListCount,
        nMaxLength-nListCount,szFormat,pArgList);
    va_end(pArgList);
    if(nListCount>(nMaxLength-1)) nListCount=nMaxLength-1;
    *(szBuf+nListCount)='\0';

    if(m_bDebug2Flag)                //如果开关打开
    {
        m_Lock.Lock();              //加锁
        {
            _Printf("%s",szBuf);    //真实执行打印
        }
        m_Lock.Unlock();            //解锁
    }
    return nListCount;              //返回长度
}

```

10.6.4 XGDebug3

```

int CTonyXiaoLog::XGDebug3(char *szFormat, ...)
{
    //这段比较经典，变参函数处理模块，不再赘述
    char szBuf[LOG_ITEM_LENGTH_MAX];
    int nMaxLength=LOG_ITEM_LENGTH_MAX;
    int nListCount=0;
    va_list pArgList;
    va_start (pArgList,szFormat);
    nListCount+=Linux_Win_vsnprintf(szBuf+nListCount,
        nMaxLength-nListCount,szFormat,pArgList);
    va_end(pArgList);
    if(nListCount>(nMaxLength-1)) nListCount=nMaxLength-1;
    *(szBuf+nListCount)='\0';

    if(m_bDebug3Flag) //如果开关打开
    {
        m_Lock.Lock(); //加锁
        {
            _Printf("%s",szBuf); //真实执行打印;
        }
        m_Lock.Unlock(); //解锁
    }
    return nListCount; //返回长度
}

```

10.6.5 _Printf

_Printf 是内部真实实现打印的函数，连贯到前几个函数，大家可以发现，笔者在设计这几个逻辑时，一来图简单，很多代码是拷贝使用，这样一目了然，不出错，二来，各个开关业务，和真实的输出分开，这体现业务层和功能层的割裂关系。

```

int CTonyXiaoLog::_Printf(char *szFormat, ...)
{
    char szTime[LOG_ITEM_LENGTH_MAX];
    char szTemp[LOG_ITEM_LENGTH_MAX];
    char szBuf[LOG_ITEM_LENGTH_MAX];
    int nMaxLength=LOG_ITEM_LENGTH_MAX;
    int nListCount=0;
    time_t t;
    struct tm *pTM=NULL;
    int nLength=0;
    //获得当前时间戳, 这在 MakeTimeString 中已经有介绍
    time(&t);
    pTM = localtime(&t);
    nLength=SafePrintf(szTemp, LOG_ITEM_LENGTH_MAX, "%s", asctime(pTM));
    szTemp[nLength-1]='\0';
    SafePrintf(szTime, LOG_ITEM_LENGTH_MAX, "[%s] ", szTemp);
    //这段比较经典, 变参函数处理模块, 不再赘述
    va_list pArgList;
    va_start (pArgList, szFormat);
    nListCount+=Linux_Win_vsnprintf(szBuf+nListCount,
        nMaxLength-nListCount, szFormat, pArgList);
    va_end(pArgList);
    if(nListCount>(nMaxLength-1)) nListCount=nMaxLength-1;
    *(szBuf+nListCount)='\0';
    //得到当前使用的文件名
    GetFileName();
    //调用 debug 的功能函数, 直接将信息输出到文件
    nListCount=dbg2file(m_szFileName, "a+", "%s%s", szTime, szBuf);
    if(m_bPrintf2ScrFlag) //如果屏幕输出开关打来
    {
        TONY_XIAO_PRINTF("%s%s", szTime, szBuf); //输出到屏幕
    }
    if(m_pInfoOutCallback) //如果拦截函数存在
    {
        //输出到拦截函数
        char szInfoOut[APP_INFO_OIT_STRING_MAX];
        SafePrintf(szInfoOut, APP_INFO_OIT_STRING_MAX,
            "%s%s", szTime, szBuf);
        m_pInfoOutCallback(szInfoOut, m_pInfoOutCallbackParam);
    }
    m_nFileSize+=nListCount; //这里很重要, 修订文件长度
    //维护模块需要这个值判定文件大小
    //是否超标
    return nListCount; //返回输出的字节数
}

```

10.6.5 XGDebug4Bin

这在 Debug 模块中已经见过, 以二进制方式输出数据块的内容, 方便调试观察。

```

void CTonyXiaoLog::_XGDebug4Bin(char* pBuffer,int nLength)
{
    m_Lock.Lock(); //加锁
    {
        GetFileName(); //获得文件名
        dbg2file4bin(m_szFileName,"a+",pBuffer,nLength); //输出到文件
        dbg_bin(pBuffer,nLength); //输出到屏幕
    }
    m_Lock.Unlock(); //解锁
}

```

10.7 Log 日志系统小结

Log 日志系统在很多场合都有使用，如果是有经验的程序员读者，也很可能在其他的各种开源、闭源系统中，看到了其内置的日志文件系统。这充分说明，日志文件，看似一个很不起眼的功能，但在商用工程中，尤其是很多哑元服务器、嵌入式设备，是必备的人机交互功能，其地位非常重要。

很多时候，Log 日志系统可以说是 Debug、性能调整的主要观察工具。甚至，在很多数据库系统中，Log 日志系统还能起到数据库重建备份的作用。

通常情况下，很多高安全性数据库集群，其数据库的备份往往是一个很大的难题。原因很简单，数据库集群，通常数据是分布在很多台服务器上存储，另外，很多时候为了业务调整需要，我们需要对数据做中间处理，定制一些纯加速作用的 Index 表，嵌入很多服务数据等。

因此，一个商用数据库服务集群，其最大的价值，往往不仅仅包含数据库中的数据，而是数据库经过长期运行的一个中间态，这包括各级服务器传输队列中的数据，各级 cache 中的数据，正在网络上传输的数据等等，所有这些数据的集合，构成了商用数据库集群的整体服务能力和最佳服务状态。换言之，即使我们备份了数据库中的数据，但如果有一天数据库崩溃，我们把数据恢复回来，重新启动集群，其运行效率在很长一段时间，可能都达不到最佳状态，因为上述很多内存中的缓存数据，是没有备份的。因此，对于高安全性服务器集群，仅仅备份数据库中的静态数据，一来不便，二来也不是很合用。

这就带来了一个话题，数据库服务集群，究竟应该备份什么？最严谨的一种做法，应该是“备份过程”，而不仅仅是备份静态数据。这就要求，显式定义每一条数据库操作信令，并且，将该数据库有史以来所有的数据库操作信令做 Log 日志记录，当某一天需要恢复时，不是渐渐单单恢复数据，而是沿着这个 Log 日志重新执行一遍，最终实现整个服务器某个时间点的服务能力的整体恢复。

这个过程看起来很吓人，因为显而易见，这些信令集合是一个只增不减的集合，会导致大量（比单纯静态数据备份大很多倍）的信息存储需求，但在一些特殊的高安全级别场合，如银行、证券、军事等方面，这是基本要求。因此，Log 日志系统看似简单，但如果深入去理解，其影响非常深远。

本章在这里也仅仅是抛砖引玉，扩展大家的思路，向大家展示开发一个高可用的 Log 日志系统，究竟需要关注哪些方面。希望各位读者通过本章的代码展示和开发思路学习，能真实理解 Log 日志系统的含义和开发原则，最终能定制出自己合用的软件模块。

第 11 章 聚合工具类

11.1 聚合工具类的类声明

11.2 聚合工具类函数说明

11.3 额外的话题：Linux 服务程序怎么写？

11.3.1 服务器的开发习惯

11.3.2 Linux 的开发习惯

11.3.3 Linux 下开发服务程序的基本需求

11.3.4 基本设计原理

11.3.5 程序实战演示

11.3.6 程序使用说明

第 11 章 聚合工具类

在前面几章，笔者为大家展示了内存池、队列、线程池、任务池以及 Log、Debug 日志系统的开发思路和详细设计方案，原则上，一个基本的 C/C++ 商用工程库，到现在已经初具规模。应用程序只要根据需要，将上述代码连接到工程中，即可实现编译连接，完成功能。

但不久前的一次开发需求，打破了笔者的看法，应用工程，仅仅依靠上述零散的类模块还不够，还需要一个有机的组织者。

需求是这样的，笔者需要开发一个 ocx 控件，这个控件将会嵌入到一个 html 页面，在 IE 浏览器中运行。但有个奇怪的限制，html 的设计者，不保证只使用我们的一个控件实例。换言之，在同一个 IE 中，可能存在我多个 ocx 的控件在同时运行。

通过前面的章节我们可以有一个结论，最起码，内存池、线程池、任务池这些模块，每个工程有一个实例就够了，一般说来，这可以使用全局对象来完成，即所有的模块都访问全局的这些对象，调用其功能。

但这种设计在上述需求中出现了问题。如果我们的 ocx 定义了一个全局的内存池，如果它是独立的 Windows 应用程序，没有问题，全局有且仅有一个内存池对象，但在 IE 内部，所有的 ocx 实例，都共享 IE 的进程私有内存空间，换言之，如果我们的 ocx 被实例化两次，则内部会出现两个全局的内存池对象，这就造成了混乱。

因此，笔者发现，工程库中的模块，原则上不允许使用全局对象直接访问，而是应该以一个统一的聚合类，将这些内容全部聚合成一个对象的功能，再在没有 ocx 中，动态实例化该聚合工具类，以后的业务层开发，全部访问这个聚合工具类指针的成员方法，方能避免上述问题。

提示：这其实说明了 Windows 下 ocx 之类的控件设计的一个禁忌，由于 ocx 的设计规范并没有禁止应用程序在每个进程中只使用一个 ocx 的实例，因此原则上，ocx 内部不允许使用全局变量，全局对象等设计，避免出现可能的“撞车”现象。

11.1 聚合工具类的类声明

首先，顾名思义，聚合工具类，是一个聚合类，即本身不实现什么具体功能，它唯一的目的，就是聚合各个功能模块，把所有模块“打包”成一个对象。

其次，这是一个工具类，即内部所有的模块指针都公开，应用层随意调用，这主要是出于简化开发起见，工程库总是在不断进步的，各个子模块可能会独立增加一些功能，也

就添加一些 api 接口，如果完全采用聚合模块，则每次子模块的修改，都需要修改这个聚合工具类，这就太麻烦了。

因此，笔者决定简单使用工具类原则，所有子模块指针全部公开，聚合层不负责处理接口函数。经过考虑，笔者把这个聚合工具类定名为 `CTonyBaseLibrary`。

另外，聚合工具类除了最基本的聚合功能外，笔者经过考虑，决定将另外一个重要功能放置其中，方便使用。

在前文介绍 `Debug` 的章节中，我们讨论过一个性能观察的话题，即 7*24 小时的服务器，有必要将一些内部动态的状态值，打印出来给程序员观察，方便进行性能和 bug 方面的跟踪。为了实现此目的，我们在很多底层类中，都设置的 `PrintInfo` 的函数接口，向屏幕打印关键变量的值。

那现在就有一个问题了，究竟谁来调用这些函数，打印中间状态值？笔者经过考虑，在聚合工具类中设置了一个线程任务 `InfoPrintTaskCallback` 来完成这个功能。下面我们来看看类声明。

11.1.1 类声明

聚合工具类由于主要起聚合作用，因此整体来说逻辑比较简单。但这里面有几个路径和回调函数的设计，可能有点费解，下面笔者将逐一说明。


```

class CTonyBaseLibrary
{
public:
    CTonyBaseLibrary(char* szAppName,           //应用名
        char* szLogPath,           //日志路径
        char* szTempPath,         //临时文件路径
        int nTaskPoolThreadMax=DEFAULT_THREAD_MAX, //任务池最大线程数
        bool bDebug2TTYFlag=true, //Debug 输出到屏幕开关
        _BASE_LIBRARY_PRINT_INFO_CALLBACK
            pPrintInfoCallback=null, //info 屏幕输出回调指针
        void* pPrintInfoCallbackParam=null, //info 回调参数指针
        _APP_INFO_OUT_CALLBACK pInfoOutCallback=null, //应用程序输出回调
        void* pInfoOutCallbackParam=null); //应用程序输出回调参数指针
    ~CTonyBaseLibrary(); //析构函数
public:
    //应用名的备份保存
    char m_szAppName[TONY_APPLICATION_NAME_SIZE];
    //日志路径
    char m_szLogPathName[TONY_APP_LOG_PATH_NAME_SIZE];
    //临时文件路径
    char m_szTempPathName[TONY_APP_TEMP_PATH_NAME_SIZE];
    //日志模块
    CTonyXiaoLog* m_pLog;
    //内存池
    CTonyMemoryPoolWithLock* m_pMemPool;
    //线程池
    CTonyXiaoTaskPool* m_pTaskPool;
    //线程池的运行体
    CTonyTaskRun* m_pTaskRun;
    //内核级 Debug, 每次运行写一个文件, 覆盖上次的
    CTonyLowDebug* m_pDebug;
private:
    //Info 打印任务
    static bool InfoPrintTaskCallback(void* pCallParam,int& nStatus);
    time_t m_tLastPrint;
    //打印信息的回调函数
    _BASE_LIBRARY_PRINT_INFO_CALLBACK m_pPrintInfoCallback;
    void* m_pPrintInfoCallbackParam; //回调函数参数指针
};

```

11.1.2 屏幕输出回调函数

笔者在聚合工具类中设计了 **InfoPrintTaskCallback** 这个线程任务, 该任务比较简单, 就是每个 2 秒 (经验值), 定时调用各个基础模块的 **PrintInfo** 函数, 向屏幕输出各种关键状态值。

但这里有一个问题, 我们知道, 聚合工具类仅仅聚合最核心的一些基础模块, 属于基本功能层, 而各个网络服务器程序, 在这些基本功能的基础上, 势必还有自己的业务层模块, 来实现各种服务业务。这些模块, 我们推论, 也可能有一些关键状态值需要输出。

如果让应用层自己来设计一个打印任务, 一来没有必要, 二来也容易引发新的 **bug**, 因此, 笔者在这里设计了一个回调逻辑, 如果应用层有自己的状态打印函数, 则可以利用这

个回调功能，实现 2 秒一次的打印逻辑。这也是帮助应用层程序员专心于自己的逻辑设计，尽量不要被不想管的基础服务功能打扰思路的一个设计。

该回调函数构型如下：

```
typedef void (* BASE_LIBRARY_PRINT_INFO_CALLBACK)(void* pCallParam);  
//static void BaseLibraryPrintInfoCallback(void* pCallParam);
```

这是一个无返回值的简单函数，聚合工具类仅仅帮助其透传一根 `void*` 的参数指针，该函数每 2 秒左右被回调一次，打印信息在基础类信息的下方。

11.1.3 应用程序输出回调函数

在前文 Log 日志管理系统这一章，我们也论述了一个问题，就是如果应用程序是一个 Windows 之类的图形界面程序，很可能不希望日志输出到控制台，而是输出到其自己设定的某个窗口中。

这需要拦截所有的 Log 日志和 Debug 输出，因此，我们设计了一个回调函数 `_APP_INFO_OUT_CALLBACK`，聚合工具类的构造函数，有责任传递这个回调函数相关的信息，到基础 Debug 和 Log 类中，因此，大家可以发现其参数中有这个指针传递要求。

```
typedef void (* _APP_INFO_OUT_CALLBACK)(char* szInfo,void* pCallParam);  
//static void ApplicationInformationOutCallback(char* szInfo,void* pCallParam);
```

11.1.4 日志文件和临时文件路径

在很多服务器开发中，日志文件和临时文件的路径是分开的。

在 Linux 系统下，一般说来，日志文件放在“/var”这个目录下，而临时文件，一般放在“/tmp”目录下。

这在程序员的角度上看，可能这个差别不明显，但从网络管理员的角度说，这两个目录有很大差异性。很多时候，一个 Linux 系统下，其他目录都是只读的，或者是链接的其他某个服务器的路径，比如“/etc”、“/sys”下的很多文件仅仅是某个网络资源的索引，而不是文件本身，这些文件都是只读的。

而一个成熟的运行态 Linux 系统，管理员明确允许应用程序任意读写的磁盘目录，很可能就只有“/var”和“/tmp”，这两者都还有差别。

1、“/var”，这里的数据一般做永久性保存，即关机不丢失，日志文件通常都放在这里，这个目录通常被管理员设置为磁盘上的一个目录，以便实现长久保存。

2、“/tmp”，原则上这是程序运行期内的临时文件，程序退出就无意义，很多网络管理员喜欢将这个目录映射到一个内存虚拟磁盘，保证高速访问特性，但不做永久性保存，关机即丢失。

因此，Linux 开发的程序员，一般需要明确界定自己的一个磁盘文件需求，是永久性存储需求，还是高速临时访问需求，并显式通过目录来界定。这不属于编程规范，但应该算作 Unix 或 Linux 程序员的一个开发惯例，必须要尊重。

原则上，这个目录定义应该被每个模块都看到，大家使用统一的参数设定。由于笔者定义本聚合工具类就是携带各种数据、指针，让每个模块都看到，任意调用执行，因此，笔者这里设计了这两个目录信息的缓冲区。

这实际上是把聚合工具类作为一个全局定义结构体使用，专门携带关键数据定义，供各个模块参考使用。

11.2 聚合工具类函数说明

聚合工具类的核心函数非常简单，就是构造函数、析构函数和信息打印线程任务函数。其中，构造函数和析构函数是最主要的功能。

11.2.1 构造函数

```
CTonyBaseLibrary::CTonyBaseLibrary(
    char* szAppName,
    char* szLogPath,
    char* szTempPath,
    int nTaskPoolThreadMax,
    bool bDebug2TTYFlag,
    _BASE_LIBRARY_PRINT_INFO_CALLBACK pPrintInfoCallback,
    void* pPrintInfoCallbackParam,
    _APP_INFO_OUT_CALLBACK pInfoOutCallback,
    void* pInfoOutCallbackParam)
{
    m_pDebug=null; //各指针变量赋初值 null
    m_pTaskPool=null; //这是预防某个初始化动作失败后
    m_pMemPool=null; //跳转，导致后续指针没有赋值
    m_pLog=null; //成为“野指针”
    //保存各个路径字符串的值
    SafeStrcpy(m_szAppName,szAppName,TONY_APPLICATION_NAME_SIZE);
    SafeStrcpy(m_szLogPathName,szLogPath,TONY_APP_LOG_PATH_NAME_SIZE);
    SafeStrcpy(m_szTempPathName,szTempPath,TONY_APP_TEMP_PATH_NAME_SIZE);
    //保存信息打印回调函数相关指针
    m_pPrintInfoCallback=pPrintInfoCallback;
    m_pPrintInfoCallbackParam=pPrintInfoCallbackParam;
    //初始化随机数种子
    srand((unsigned int)time(NULL));
}
```

学习过 Windows 网络编程的读者可能知道，Windows 的 Socket 相关 api 和 Unix、Linux 的都不一样，并不是完全执行伯克利规范，而是自己另外做了一套规范，仅仅是部分形似而已。

因此，Windows 的 Socket 开发有一些怪癖，比如要求应用程序启动时，必须初始化 Socket，退出时必须关闭，此处的代码就是初始化动作。下面析构造函数中还有结束 Socket 的动作。

```
#ifdef WIN32
{
    //注意，大括号限定作用域，可以临时定义变量
    m_bSocketInitFlag=false;
    WORD wVersionRequested;
    int err;
    wVersionRequested = MAKEWORD( 2, 2 );
    err = WSStartup( wVersionRequested, &m_wsaData );
    if ( err != 0 )
    {
        TONY_DEBUG("Socket init fail!\n");
    }
    else m_bSocketInitFlag=true;
}
#else // Non-Windows
#endif
```

第一个动作，初始化 Debug 对象，万事从 Debug 开始

聚合工具类的析构函数非常重要，这是保证所有基础模块能按顺序，安全析构释放的关键。

```
CTonyBaseLibrary::~CTonyBaseLibrary()
{
    //笔者习惯，这是应用程序逻辑开始退出的标志，相对于构造函数中的输出
    TONY_DEBUG("<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n");
    //这是一个技巧，退出时，笔者喜欢按照功能模块的划分，打印一些数字
    //当某个模块线程出现 bug，其退出逻辑很可能受到影响，导致退出时挂死
    //此时只要简单观察数字打到几，就可以轻松判定是下面哪一步出现了问题，方便 debug
    TONY_XIAO_PRINTF("1\n");
    //设置内存池的关闭标志，内存块的 Free 动作将直接释放，不再推入内存栈，加快程序运行
    m_pMemPool->SetCloseFlag();
    TONY_XIAO_PRINTF("2\n");
```

```
    TONY_XIAO_PRINTF("3\n");
    if(m_pTaskRun)                                //退出线程池运行体，注意其中反注册动作
    {
        m_pMemPool->UnRegister(m_pTaskRun);
        delete m_pTaskRun;
        m_pTaskRun=null;
    }
    TONY_XIAO_PRINTF("4\n");
```

```
    TONY_XIAO_PRINTF("5\n");
    if(m_pTaskPool)                                //退出线程池，注意其中反注册动作
    {
        m_pMemPool->UnRegister(m_pTaskPool);
        delete m_pTaskPool;
        m_pTaskPool=null;
    }
```

```
    TONY_XIAO_PRINTF("6\n");
    if(m_pLog)                                      //退出 Log 日志模块，注意其中反注册动作
    {
        m_pMemPool->UnRegister(m_pLog);
        delete m_pLog;
        m_pLog=null;
    }
```

```
    TONY_XIAO_PRINTF("7\n");
    if(m_pMemPool)                                //退出内存池
    {
        delete m_pMemPool;
        m_pMemPool=null;
    }
```

```

TONY_XIAO_PRINTF("8\n");
TONY_DEBUG("Bye World!\n");           //笔者习惯，“再见，世界”
TONY_DEBUG("-----\n");
if(m_pDebug)                           //退出 Debug 对象
{
    delete m_pDebug;
    m_pDebug=null;
}
TONY_XIAO_PRINTF("9\n");

```

这里就是 Windows 下退出 Socket 的代码

```

#ifdef WIN32
    if(m_bSocketInitFlag)
    {
        if ( LOBYTE( m_wsaData.wVersion ) != 2 ||
            HIBYTE( m_wsaData.wVersion ) != 2 )
        {
            WSACleanup( );
        }
        m_bSocketInitFlag=false;
    }
#else // Non-Windows
#endif
}

```

11.2.3 InfoPrintTaskCallback

信息打印任务回调函数，是实现所有动态状态输出的关键。笔者一般定义 2 秒输出一次，不过，这里也是使用宏表示的时间。

```

#define MAIN_LOOP_DELAY 2    //2 秒一次 main loop
bool CTonyBaseLibrary::InfoPrintTaskCallback(void* pCallParam,int& nStatus)
{
    //强制指针类型转换，获得本对象指针
    CTonyBaseLibrary* pThis=(CTonyBaseLibrary*)pCallParam;
    //检测是否已经等到 2 秒，否则返回
    if(TimeIsUp(pThis->m_tLastPrint,MAIN_LOOP_DELAY))
    {
        TimeSetNow(pThis->m_tLastPrint);           //更新当前时间
        TONY_XIAO_PRINTF("*****\n");
        pThis->m_pTaskPool->PrintInfo();           //打印任务池信息
        pThis->m_pTaskRun->PrintInfo();           //打印任务池运行体信息
        pThis->m_pMemPool->PrintInfo();           //打印内存池信息
        if(pThis->m_pPrintInfoCallback)           //回调应用层打印函数
            pThis->m_pPrintInfoCallback(pThis->m_pPrintInfoCallbackParam);
        TONY_XIAO_PRINTF("*****\n");
        TONY_XIAO_PRINTF("\n");
    }
    return true;                                   //返回真，永远循环
}

```

请注意其中 TimeIsUp 的定义。这里我们复习一下。

```
#define TimeSetNow(t) time(&t)
inline bool TimeIsUp(time_t tLast, long lMax)
{
    time_t tNow;
    TimeSetNow(tNow);
    long lDeltaT=(long)tNow-(long)tLast;
    if(lMax<=lDeltaT) return true;
    else return false;
}
```

这里面大家可能注意到一个重要问题，笔者使用了一个类成员变量 `m_tLastPrint` 来做单独的计时（此处相当于全局变量，静态变量，即变量生命周期长于函数生命周期）。这里有个很重要的细节需要大家关注。

任务池的任务，虽然我们已经论述了，一定会被重复调用，不断循环，但是，请注意，任务池、操作系统并不对重复调用的时间间隔做保证，因此，不能简单以被调用多少次来推算真实的时间间隔。

因此，本函数中，当我们需要精确计时 2 秒时，就不能简单通过循环计数器来推算时间，而必须以标准 C 的时间函数来检测时间差。如上面的 `TimeIsUp`。

这里还可以给大家多分享一点知识：线程中的睡眠，永远是不准确的，受很多因素制约。比如我们希望睡眠 10ms，`Sleep(10)`，但是，我们可以设想一下，操作系统是如何调用我们的线程的？

通常，操作系统会在任务链表中不断遍历循环，这循环一周的时间，取决于很多条件：

- 1、任务队列很长，操作系统遍历一轮，可能需要几百个毫秒，这是一个影响条件。
- 2、任务优先级，我们的应用程序，可能优先级很低，别的线程调用几次，才会调用我们 1 次，这也是一个影响条件。

- 3、每个线程循环是否都睡眠了，这不仅仅是我们的程序，也要看同时运行的其他程序，如果某一个程序写得不够好，大循环中忘了睡眠，它每一次都会把它的时间片用完才被强制打断，则整个系统的效率会很低，此时，遍历任务链表一遍的时间也会变长。

- 4、操作系统自己的任务影响，多任务操作系统，不管是 Windows 还是 Linux，都有一些系统级的守候任务，如虚拟缓存磁盘交换，文件 cache 的磁盘写入等，这些任务可能会因为某个时刻的条件满足而启动，这是随机的，因此，我们获得时间片的机会也是随机的。

由于有上述种种限制，因此，当我们 `Sleep(10)` 时，一般都不是刚好 10 毫秒就会苏醒，而是大于这个数值，根据笔者观察，下述循环的周期，一般是 2 毫秒，即苏醒一次的代价，差不多 1ms 左右。

```
void Func(void)
{
    while(1)                //死循环
    {
        Sleep(1);           //每循环睡眠 1
    }
}
```

11.3 额外的话题：Linux 服务程序怎么写？

在笔者多年的开发生涯中，发现一个很有趣的规律，工程实战和程序开发学习之间，有个最大的差别，就是细节。

我们在学校里面学习程序开发，不可避免地会做很多练习题，并形成一些基本的开发思路，很多年轻的朋友应聘工作时，就以此为项目经验向面试官介绍。

但通常大家会发现，很多时候面试官并不看重这类经验，原因很简单，练习题就是练习题，永远无法替代商用工程项目的开发经验。二者最关键的差异性，就在于对细节的把握程度大相径庭。

比如本小节的这个话题，就属于典型的细节讨论，甚至，这是一个不是话题的话题，但是，这个细节如果解决不好，我们写出来的，永远是习作程序，不是商用服务器软件。

11.3.1 服务器的开发习惯

目前，业界主流的服务器开发和运维平台，是 Linux，笔者在本书中讨论的很多服务器话题，虽然都标明了跨平台兼容，但主要的目标运行平台，还是 Linux。

不过，在讨论 Linux 服务程序开发习惯之前，我们还是先讨论一下大家熟悉的 Windows 系统。准确的讲，Windows 并不是一个很适合做服务器的操作系统，即使微软公司推出了这样那样的服务器版 Windows。

开个玩笑讲，用 Windows 做服务器操作系统，一般来说，只有两个问题不太合适：“这也不合适，那也不合适”。

所有的 C 或 C++ 程序员，写出的第一个程序，可能就是“Hello World”了，这说明一个小小的问题，从第一天写程序开始，我们就在和控制台打交道。而抽象来说，字符型的控制台 CUI 界面和 Windows 下的 GUI 界面，从本质上讲并无不同，都是作为用户的“人”和实现计算的“计算机”之间的一种交互手段而已，都叫做“UI (User Interface、用户界面)”

但上述开发习惯，自然给程序员带来一种不太好的思维惯性，即程序是围绕 UI 在工作的，这中思维到了微软公司的 Windows 下，得到了前所未有的发挥。

只要用纯 C 方式写过 Windows 程序的读者应该知道，Windows 程序一开始，从 WinMain 进入，一般就是一大段启动代码，初始化并显示出一个窗口，随之就会进入一个大的消息死循环，不断检索各种事件（键盘、消息、屏幕重绘等等），然后执行，直到收到退出事件，程序结束。

这意味着什么？是不是意味着，Windows 程序的开发思维，从一开始就是围绕 UI 在进行，屏幕的输出、和键盘鼠标的输入被视为一个 Windows 程序最基本的功能。如下图所示：



图 11.1 Windows 程序观点

我们从图上可以看到，Windows 程序开发的核心思维，是以 UI 为中心，所有的计算功能，包括业务计算，都是零散的子功能，是等待 UI 事件发生，而被调用的对象。

这在以终端客户为主要服务对象，以 GUI 为卖点的 Windows 中，这种程序设计模型，本来也无可厚非。因为大多数 Windows 应用程序，属于典型的人机交互密集型的 UI 服务程序，基本上属于“用户点击哪个功能，就执行哪个模块”，一起动作的起点，是用户的输入。

Windows 系统的基本优化核心也在于此，各个系统模块，都以 UI 的顺畅执行为基本优化目标，这贯穿了 Windows 这个操作系统整体的开发思维。

但是，这种模型在商用工程的服务器开发中，就出现了问题。

我们知道，通常服务器是运行在后台，很多时候，服务器没有本地用户交互的需求，一般都哑元运行，管理员通常只有在服务列表、进程列表，或日志文件中，才能感受到服务程序的运行。

但这是不是意味着服务器就没有 UI 了呢？笔者认为不是。就笔者看来，UI 就是一套输入输出界面，其存在的意义就是为应用程序提供外部的数据输入，以及把业务上需要输出的信息（文本或二进制）输出到合适的设备上，仅此而已。

从这个意义上讲，UI 抽象了看，就是应用程序的输入和输出流设备。而服务器程序通常情况下是用于网络服务，其业务数据，“**来自于网络，输出到网络**”。因此，它的 PI（Programmer Interface，程序界面）可以认为是它的另外一套 UI，前文所述的 api 和 npi，都是这个含义，这是抽象理解。

因此，服务器可以说，是以网络为 UI 的一套应用程序。这时候，就体现出 Linux 和 Unix 下程序设计思路的先进性，在 Linux 看来，所有的程序，是围绕其主营业务服务的，UI、npi、api，仅仅是接口，是不保证服务的非关键因素，如下图所示：



图 11.2: Linux、Unix 程序观点

如图，在 Linux 和 Unix 下，服务程序提供 UI、api 和 npi 三种基本的接口格式，一般都是“请求----响应”型工作模型，在 Linux 应用程序或服务器看来，一个请求，无论是来

自于 UI 还是其他两种接口，都是一种抽象的请求格式，程序的回应也返回相同抽象格式的数据，各个接口自行解释成自己的客户需要的数据格式。

1、本地用户，可以通过控制台的 CUI 或者 XWindow 的 GUI 方式，提出请求，访问服务，并获得回应。

2、本机运行的其他运行模块，可以通过 api 提出请求，访问服务获得回应。

3、网络上的其他角色，可以通过 npi，远程提出请求，获得回应。

此时，我们回头再看看 Windows 下的程序设计观点，就可以看出问题，Windows 程序至关重要的消息循环，其实抽象来看，仅仅是一层 UI 行为，是与各个应用程序自身业务逻辑无关的一个问题，在 Linux 这张图上，它仅仅是 GUI 这个模块中的一部分而已。二者的程序抽象性显然不同，并且，显然 Linux 下的抽象性更好。

因此，Linux 服务器程序员，一般很少关注 UI 特性，其主要的精力，都是放在服务器自身的主营业务上，几乎所有的设计与实施，都是以业务优化为前提，这样开发出来的服务器程序，其效率显然比掺杂了大量 UI 考虑的 Windows 程序效率要高。

举个不太恰当的例子，大家看到很多嵌入式服务器应用，如 ADSL Modem，家用 Soho 路由器、交换机，机顶盒设备，基本上都是基于 Linux 开发，而很少见到 Windows 的嵌入式版本 WinCE 开发的产品，从纯技术角度上考虑，Linux 的精炼，短小和高效，应该是关键因素。

上述原因，仅仅是从程序设计世界观的角度，探讨了 Linux 下开发服务器，相对 Windows 系统的优劣性。不过，请注意，通常情况下，这不是主要原因，因为很多时候，效率不是商用化系统的唯一选择。

Linux 成为主流的服务器运行平台，最根本的原因还是它免费，相对 Windows 服务器平台动辄几百美金的授权费用，开源的 Linux 系统部署费用基本上为 0，这直接影响了很多企业的平台选型。

因此，笔者在本章中特别增加了本小节的话题，即在 Linux 下开发服务器程序，应该关注的一些基本细节。

提示：Windows 系统其实提供了专门的“服务（Service）”开发框架，有单独的运行接口，该框架也并不如本文所说的，围绕 UI 运作。因此，本文的提法，可能有点偏颇。不过，Windows 下开发“服务”的应用，毕竟是少数，大多数 Windows 程序员，基于 Win32API 和 MFC 养成的习惯，仍然是 UI 为主，笔者认为这不太好，可能会影响程序员自身的发展，因此本文刻意把这种思路拿出来做比较，请各位读者仔细思考其中深意。

11.3.2 Linux 的开发习惯

虽然程序开发本身是一种抽象，但各个操作系统由于其历史的渊源，通常还是有一些约定俗成的开发习惯。前文所述的关于 Linux 下“/var”和“/tmp”这两个目录的定义，就是这类“风俗”。

通常情况下，Linux 系统并没有太刻意去区分服务和应用程序的差异性，这和 Windows 系统有显著差别。在 Windows 下，一个 7*24 小时的服务程序，一般建议开发成一种独特的应用----“服务”，这中服务程序的开发，和普通应用程序有显著差异。

就笔者理解，Windows 服务程序与应用程序最核心的差异就是弱化、忽略了前文所述的 UI 相关的消息循环，而代之以一套服务事件机制，该机制的效率比普通应用程序要高很多。同时，Windows 的服务进程，优先级天生就比应用程序高很多。

不过在 Linux 下，虽然也有进程、线程优先级的设计，但通常开发时，程序员较少去关心这些细节，客观地讲，这属于网络管理员的工作范畴，一个程序的具体运营使用优先级，一般是系统设计师给出建议，网络管理员实施部署。

在 Linux 下，服务程序的开发和应用程序开发，大多数时候是相同的，都是以一个 main 函数开始运行，直到结束。但也有一些不同点。

但就运营而言，服务器和应用程序最核心的不同点在于，服务器，一般建议是在后台运行，从启动脚本中启动，而不是管理员手工从控制台键入文件名开始运行，甚至可以这么说，Linux 下程序的运行，与控制台无必然联系，一个程序，可以由用户手动启动，也可以由 shell 脚本来启动，运行结果都一样。

这点已经可以看出 Linux 的程序观点，就是上文所述的以业务为核心。UI 不是 Linux 程序运行的必要条件。

但这往往会带来一个管理上的问题，一个服务器在后台运行，用户在控制台，通常无法直接与其做键盘交互，如果中途希望中断一下，做一些维护工作，只能用 kill 命令直接杀掉进程。但这显然是一种很粗暴的行为，不值得提倡。

我们知道，程序运行时，有很多二元动作，很多都是与系统资源有关，通常情况下，内存申请，socket 使用等系统资源，是与进程绑定的，当进程死亡，资源无需显式关闭，自然会被系统回收，因此，kill 进程通常情况下，对二元动作的影响，没有程序内部资源泄漏那么大。

但是，这也不能掉以轻心，我们知道，很多时候操作系统对磁盘文件的操作，是有缓存的，在商用服务器程序中，内存缓存技术也大量存在和使用，我们前文所述的内存队列，就是典型的例子。由于这些信令退出时不允许丢失，必须保存到磁盘文件上，这其实也构成了一个二元逻辑，即运行期处理不完的数据，退出时必须保存到磁盘文件或数据库，下次启动读入，继续处理。

因此，几乎所有的服务器程序，都不允许上述“粗暴”退出，以免丢失内存中的信令数据。这就要求，网络服务器的开发，通常要求“温和”退出，即类似线程的处理原则，严禁在外部强行 kill，必须以一定的通知机制，通知服务程序“自己”退出来，并且，一定还要有一套机制等待服务程序退出完毕，即所有应保存的数据都确实保存到磁盘文件中以后，系统才能安全退出。

这必须做一些特殊的设计工作。而且，这是一些很细节的设计，但如果不关注这一点的话，我们的程序，永远只是“玩具”，而不是“工具”。

11.3.3 Linux 下开发服务程序的基本需求

简单说，Linux 下 7*24 小时长期服务程序，原则上应该有如下基本特征：

1、如果是控制台运行，应该有人机交互界面，即允许管理员“手动”、“温和”退出程序。

2、如果是后台运行，也应该提供一个命令，允许管理员在本机，通过命令通知服务器，“温和”退出程序。

3、这个退出命令，应该允许放到 shell 脚本中运行，以便在开关机时自动运行。

4、“温和”退出，还必须有个显式的提示，提醒调用者（主要是 shell 脚本程序），已经退出完成，可以执行下一步动作，换句话说，服务器进程和退出脚本是异步运行的，必须给出一个同步信号，将脚本做一点“悬挂”或“阻塞”同步，否则的话，系统的退出脚本极有可能在服务器尚未完全保存好数据时关闭系统，最终还是丢失数据。这很类似线程安全锁中 nThreadCount 的设计。

当然，这里讲的都是本机操作，如果有需求的话，也可以考虑使用 socket 连接构建一套远程的关闭通知系统，实现远程起停服务器。不过，这不是本书讨论的主要话题，因此笔者在此不再赘述。

11.3.4 基本设计原理

如果要实现上述需求，我们主要要解决下面三方面的问题：

11.3.4.1 控制台命令控制

首先声明，虽然 C 语言提供了 `scanf` 函数，但是商用工程一般不建议使用。因为它太不安全了，缺乏必要的防护机制，很容易由于用户错误的超长输入，导致缓冲区溢出，进而导致程序崩溃。

在 Windows 下，笔者一般喜欢使用自己编写的一套 `SafeScanf` 的函数，如下：

```
#ifdef WIN32    //仅在Windows下有效
int SafeScanf(char* szBuffer,    //给定输入缓冲区指针
               int nLength)      //给定输入缓冲区长度
{
    char ch;                    //最近一次输入的字符
    int i=0;                    //缓冲区指针
    while(1)
    {
        if(kbhit())             //有无按键被按下
        {
            ch=getch();          //有的话，使用 getch 函数获得键盘输入的字符
            if('\r'==ch) break;  //回车键跳出循环
            if('\n'==ch) break;
            if(nLength>(i+1))    //长度不超限的话执行逻辑
            {                    //i+1 是为了给最后的'\0'留出位置
                if(32>ch)        //过滤掉控制字符
                    goto CTonyBaseLibrary_SafeScanf_Loop;
                else if(32==ch)
                {                //过滤掉命令输入前的空格
                    if(!i)       //如" exit now", 会过滤成"exit now"
                        goto CTonyBaseLibrary_SafeScanf_Loop;
                }
                TONY_XIAO_PRINTF("%c",ch); //将有效字符打印到屏幕
                *(szBuffer+i)=ch; //有效字符添加到缓冲区
                i++;              //缓冲区指针+1
            }
        }
        CTonyBaseLibrary_SafeScanf_Loop:
        continue;
    }
    Sleep(50);                  //任何循环都应该有 Sleep，相对键盘输入而言
                                //50ms 的循环检测间隔，绰绰有余
    TONY_XIAO_PRINTF("\n");    //跳出后，输出回车符
    *(szBuffer+i]='\0';        //字符串后面添加'\0'
    return i;                  //返回有效字符数 (strlen 的长度，不包含'\0')
}
#else // not WIN32
#endif
```

通常 Windows 下开发控制台服务程序时，笔者在 `main` 的主循环中会调用上述函数，一般以 “`exit`” 字符串作为退出标志，“温和” 退出程序。

但在 Linux 下这并不容易，gcc 在 CUI 控制台模式开发时，一般都不具备用户键盘输入的基本调用，简单说就是 `getch` 这个函数失效，gcc 的基本库没有提供。必须使用一个 `curses` 库才能实现。

但这很麻烦，一来是这个库是基于一种文本窗口的概念在管理控制台，需要做很多初始化和收尾等二元动作，这引入了额外的工作量。另外，基于前文所述理由，笔者一般不太愿意在服务器程序中引入太多业务无关的 UI 库。因此，上述 `SafeScanf` 程序笔者一般不在 Linux 下使用。

不过，“上帝为我们关上一扇门，就会为我们打开一扇窗”，在 Linux 下，有另外一个系统级功能调用我们可以借用，“`ctrl-c`”，这是 Linux 下常规的控制台中断进程的键盘操作，不过，通常情况下，这个功能很不好，它是一种“粗暴”打断，会导致数据丢失。

但好在 Linux 为我们提供了拦截这个信号的方法，因此，我们可以在自己的程序中，拦截这个信号，再通过一定的设计，实现温和退出。因此，笔者的 Linux 服务器程序，通常使用“`ctrl-c`”来做“手动温和”退出。

11.3.4.2 本机跨进程通知

Windows 下所有的进程间信息传递，基本上都是事件，Linux 这点和 Windows 有较大差异，Linux 一般使用信号量。

前面的“`ctrl-c`”其实就是由键盘引发的一个信号量，该信号量被传递给当前前台进程，如果我们拦截了该信号量，也就实现了“手动温和”退出。这个信号量，gcc 提供的宏叫做 `SIGINT`

但跨进程“通知退出”和这个还不太一样，不过原理差不多，Linux 和 Unix 一般约定了一个 `SIGTERM` 作为标准退出信号，也就是说，我们只要通过程序，向一个进程发出这个 `SIGTERM` 信号，那个程序就会被退出。

而如果我们要实现“通知温和”退出，就很简单了，拦截 `SIGTERM` 信号即可。

当然，这里面还有个很重要的细节，跨进程退出发信号，我们首先要知道这个进程的 ID (PID)，这在 Linux 下是动态的，必须通过查询知道。

为了让程序自动获得这一点，我们可以在我们的服务程序中，主动检测出自己的 PID，然后把它写到一个文件中，退出程序可以使用这个文件获得 PID，然后发出消息，通知对方退出。

在 Linux 下，这类 PID 描述文件也有规定，一般叫做“`xxxx.pid`”，放在“`/var/run`”这个目录下。如“`/var/run/my_server.pid`”。

11.3.4.3 退出信号（通知关闭者）

当我们选择了上述跨进程退出方案后，退出同步信号也变得很简单了。Linux 的管理，一个进程存在，它的 `pid` 文件也应该存在，反之，进程死亡，`pid` 文件应该随之删除。按照这个设计要求，我们的服务器程序退出时，必须删除 `pid` 文件。

那我们的退出程序原则上在发出退出信号后，可以以一个守候循环来检测这个 `pid` 文件是否存在，如果存在，则表示服务器没有退出完毕，必须等待，否则就可以返回，进行下一步了。

这就实现了类似 `nThreadCount` 的“退出后通知”功能。

11.3.5 程序实战演示

上述原理说起来很复杂，但程序实做起来其实还是很简单的。下面，笔者利用一个实例，为大家展示这一技巧。

11.3.5.1 服务器安全退出机制逻辑

由于本书所介绍的所有技巧，均基于 Windows 和 Linux 跨平台开发，因此，在本例中的代码，也可以在 Windows 下编译通过。

1、如果是 Windows 下工作，程序表现为一个控制台程序，启动后进入死循环，开始服务。直到用户从键盘输入“exit”，并回车，程序安全退出。但 Windows 下无下述功能。

2、Linux 下同 Windows 下类似，也是启动后进入死循环，开始服务，直到用户从键盘输入“ctrl-c”，程序安全退出。

3、如果 Linux 下程序运行于后台，则可以通过调用“程序名+-q”，退出在本机运行的本程序的另外一个实例。如一个服务程序名为“MyService”，并已经在后台运行，则用户可以从任何一个控制台运行“MyService -q”，即可退出前面运行的实例。

4、利用本机制，可以实现互斥运行，即一个“MyService”已经运行时，用户再次输出“MyService”运行新的实例，程序会自动发现已经有实例运行，并及时退出。这很重要，很多网络服务程序都是监听固定端口，由于 TCP 的端口具有唯一性，因此，多重运行没有意义，反而容易造成系统故障。

5、上述功能主要是通过监视 pid 文件实现的，但有时候，也可能因为某一次错误的崩溃退出，导致磁盘上的 pid 文件没有删除，因此，每次运行“MyService”时，程序会误以为“MyService.pid”仍然存在，因此拒绝运行。为解决这一问题，笔者设计“MyService -c”来清除上一次错误的 pid 文件，保证程序可以顺利进行。

11.3.5.2 Pid 相关函数

pid 相关函数是本节最重要的内容，基本上所有的功能实现，都是基于 pid 文件而实现的。我们先看其定义，为了保证 Windows 下也能编译通过，因此这里使用了条件编译，为 Windows 程序也提供了一个 pid 文件。

```
#ifndef WIN32
#define WB_RELAYER_PID_FILENAME "c:/MyService.pid"
#else // not WIN32
#define WB_RELAYER_PID_FILENAME "/var/run/MyService.pid"
#endif //WIN32
CMbool g_bMainExitFlag; //这是全局退出标志变量
```

首先，我们要有一个函数，来保存 Linux 下运行进程的 pid。

```
static bool SaveMyPid(void) //可能会失败
{
    bool bRet=false; //准备返回值
    FILE* fp=null; //准备文件指针
#ifdef WIN32
    int pid=65535; //Windows 下，pid 值默认为 65535，仅仅是起示意作用。
#else // not WIN32
    pid_t pid=getpid(); //调用 Linux 下的 getpid 函数，获得本进程 pid
#endif //WIN32
    fp=fopen(WB_RELAYER_PID_FILENAME,"wt"); //打开 pid 文件
    if(!fp) goto SaveMyPid_End_Process; //出错返回 false
    fprintf(fp,"%d\n",pid); //写入 pid 信息
    fclose(fp); //关闭文件
    bRet=true; //返回 true
SaveMyPid_End_Process:
    return bRet;
}
```

当然，既然我们提供了创建 pid 文件的功能，也需要提供删除 pid 文件的功能。

```

void DeleteMyPid(void)
{
    //调用 Debug 模块中的接口删除文件
    CTonyLowDebug::DeleteAFile(WB_RELAYER_PID_FILENAME);
}

```

当我们的程序以“-q”方式运行，即准备退出另外一个实例时，首先应该读取该实例的 pid 文件，以便实现 Linux 跨进程通信退出。注意，本函数一定要在本实例的 SaveMyPid 前调用，否则看到的就是本实例的 pid 了。下文的代码可以描述这个细节。

```

//读取一个远端进程的 pid，成功，返回 pid 值，失败，返回-1
static int GetRemotePid(void)
{
    char* szTemp=null;                //临时指针
    char szBuffer[256];               //缓冲区
    int nRet=-1;                      //初始化返回值
    FILE* fp=null;                   //文件指针
    fp=fopen(WB_RELAYER_PID_FILENAME,"rt"); //打开文件，由于前文宏定义一致，
                                        //因此，所有实例打开的 pid 文件，
                                        //是一致的。不会张冠李戴。

    if(!fp) goto GetRemotePid_End_Process; //打开失败，返回-1，这说明远端
                                        //实例很可能没有运行。

    szTemp=fgets(szBuffer,256,fp);     //利用 fgets 获得 pid 文件的内容
    if(!szTemp) goto GetRemotePid_End_Process; //失败返回-1
    nRet=atoi(szTemp);               //atoi，获得 pid 值
GetRemotePid_End_Process:
    return nRet;                      //返回
}

```

这里还有一个辅助函数，仅仅返回 pid 存在与否。

```

static bool IHaveRunning()
{
    return (0<=GetRemotePid());      //不是-1 就是有 pid。
}

```

11.3.5.3 Linux 中断拦截函数

为了实现本小节的目标，我们的 Linux 服务程序至少需要拦截两个终端，“ctrl-c”的 SIGINT 和标准退出信号的 SIGTERM。这里，我们需要先准备两个中断处理函数，使拦截到的中断，有落点。

```

//static function CTRL+C SIGINT 信号屏蔽函数
static void _DisableCtrlcFunc(int s)
{
    TONY_XIAO_PRINTF("Get ctrl-c single, exit!\n");
    g_bMainExitFlag.Set(true);      //设置全局退出标志
    return;
}
//static function 中断 15 SIGTERM 信号屏蔽函数
static void _RemoteExitFunc(int s)
{
    TONY_XIAO_PRINTF("Get remote exit single, I will exit!\n");
    g_bMainExitFlag.Set(true);      //设置全局退出标志
}

```

11.3.5.4 主循环逻辑

这里面其实有一个不是话题的话题，在笔者刚刚开始学习多线程开发时，曾经犯过一阵迷糊：“线程就是执行生命体，就可以完成功能，那么，当所有的业务都用线程完成的时候，主线程做什么？”

大家不要笑，当时笔者这点确实没有想通。但后来，在项目团队开发中，居然也有年轻的程序员问笔者这个问题，笔者就觉得不好笑了。看来迷糊的人不止笔者一个。

仔细想想，这确实是一个问题，当我们以一套复杂的逻辑，按步骤启动各个服务线程，开始正常的服务业务，main 函数作为主线程，确实应该做点事情，否则，它马上退出，整个进程就退出了，所谓的业务也开展不起来。

因此，main 函数中至少应该有一个主循环，系统的退出，通常是以跳出这个主循环，main 函数执行结束代码，最后退出的。经过仔细思考，笔者决定还是给这个主循环安排点事情做，就是在 Windows 下，接收并响应键盘输入，以便退出。Linux 下不需要这么考虑，因为都是中断运行，相当于有另外一个线程在修改 g_bMainExitFlag 这个变量的值，因此，Linux 下这就是简单一个死循环。

因此，笔者的主循环，通常是如下逻辑：

```
void main(int argc, char* argv[])
{
    //...初始化代码
    while(!g_bMainExitFlag.Get()) //检查主退出开关，Linux 下用
    {
#ifdef WIN32
        SafeScanf(szBuffer, 256); //调用前文的 SafeScanf，获得键盘输入
        if(0==strcmp("exit", szBuffer)) //如果是“exit”，就跳出主循环，退出
        {
            break;
        }
        else //命令错了，打印提示信息
            TONY_XIAO_PRINTF("\'%s\' is not command!\n", szBuffer);
#else // not WIN32 //注意，Linux 下不做什么事
//仅仅等待中断修改 g_bMainExitFlag 的值
        Sleep(50); //main 主循环也是线程，也要 Sleep
    }
    //...结束代码
}
```

11.3.5.5 main 函数

通过上述的准备工作，我们基本上可以确定，我们的程序一般有三种运行模式：

- 1、普通运行模式，这是实现正常的服务功能。
- 2、远端实例退出模式，这是使用-q 参数调用，发出退出信号，通知远端正在普通运行模式的实例退出。
- 3、清空 pid 文件模式，这是使用-c 参数调用，清除错误退出导致的残留 pid 文件，以便下次可以进入普通运行模式。

我们先来看看 main 函数怎么书写：


```

int main(int argc, char* argv[])
{
    if(1==argc)                                //如果没有参数，表示普通运行模式
    {
        main_run_normal(argc,argv);
    }
    else if(0==strcmp("-q",argv[1]))           //如果带-q 参数，进入远端实例退出模式
    {
        main_exit_remote(argc,argv);
    }
    else if(0==strcmp("-c",argv[1]))           //如果带-c 参数，进入清空 pid 文件模式
    {
        main_clean_run_flag(argc,argv);
    }
    else                                         //其他的错误参数，打印说明文字
    {
        TONY_XIAO_PRINTF("MyService by Tony.Xiao\n");
        TONY_XIAO_PRINTF(" MyService : run, if another process is running, will
break!\n");
        TONY_XIAO_PRINTF(" MyService -q : exit remote running process!\n");
        TONY_XIAO_PRINTF(" MyService -c : clean run flag\n");
        TONY_XIAO_PRINTF(" MyService -i : help\n");
        TONY_XIAO_PRINTF("\n");
    }
    return 0;
}

```

这里假定我们的示例工程名为 MyService。有一个细节请注意，Linux 下，GUI 不是必须的，很多运营的 Linux 服务器也不会安装 XWindow 系统，所有的操作仅仅通过文本控制台来完成。

因此，请大家不要奢望在 Linux 下可以自如地使用中文。所有的中文字符，原则上都会被显示成乱码。因此，所有需要打印到屏幕上的说明文字，请全部使用英文，否则管理员无法看懂。

笔者和项目团队一般有个不成文的约定：“所有的注释，建议使用中文，因为毕竟大多数时候是中國人在看，所有的打印文字，都使用英文表意，另外，严禁使用汉语拼音。”

11.3.5.6 普通运行模式

普通运行模式最复杂，因为它一旦进入，首先要做互斥性判断，避免运行两个实例，另外，还要实现原有 main 所有的初始化和退出逻辑。整个服务程序的主要功能，就在这个函数中实现。

```

int main_run_normal(int argc, char* argv[])
{
    char szBuffer[256];                //准备键盘输入缓冲区
    if(IHaveRunning())                 //首先做互斥性判断
    {
        //如果发现已经有实例在运行，打印说明信息，退出。
        TONY_XIAO_PRINTF("I found another process is been running!\n");
        TONY_XIAO_PRINTF("So,I will break!\n");
        TONY_XIAO_PRINTF("If you are sure there is not another \"MyService\" is
been running!\n"); //请注意这句：“如果您确定没有另外一个实例在运行”
        TONY_XIAO_PRINTF("Maybe last running is broken by some mistake!\n");
        //那可能最后一次退出是因为某种错误（非法退出）
        TONY_XIAO_PRINTF("Please use \"MyService -c\" to clean run flag, and run
me again!\n"); //请使用 MyService -c 清除上次的（pid 文件）标志。
        return 0;
    }
    //请注意先后顺序，先做互斥性判断，检查上次的 pid，再保存本实例的 pid
    if(!SaveMyPid())
    {
        //保存失败，原因很多，磁盘满了，另外一个实例正在写这个文件等等
        //但都视为非法状态，打印报警信息，退出。
        TONY_XIAO_PRINTF("Save My pid fail,please check your disk!\n");
        return 0;
    }
}

```

下面进入初始化代码阶段，由于笔者的聚合工具类在很多场合都有使用，因此，这里直接给出聚合工具类的初始化方法。

```

{
#ifdef WIN32    //Win32 模式
    m_pBaseLibrary=new CTonyBaseLibrary("wb_relayer",
        "",                //日志路径为空（当前目录）
        "",                //临时文件路径为空（当前目录）
        RELAYER_DEFAULT_THREAD_MAX,
        true,
        null);            //不要打印回调
#else // not WIN32 (Linux 模式)
    //拦截“ctrl-c”的 SIGINT 信号
    if(signal(SIGINT,_DisableCtrlcFunc)==SIG_ERR)
    {
        TONY_XIAO_PRINTF("can't catch SIGINT,ctrl+c cache fail\n");
        exit(1);
    }
    else TONY_XIAO_PRINTF("ctrl+c is set to safe exit!\n");
    //拦截“中断 15”，标准退出信号 SIGTERM
    if(signal(SIGTERM,_RemoteExitFunc)==SIG_ERR)
    {
        TONY_XIAO_PRINTF("can't catch SIGTERM,TERM is not catch\n");
        exit(1);
    }
    else TONY_XIAO_PRINTF("TERM is catch!\n");
    m_pBaseLibrary=new CTonyBaseLibrary("wb_relayer",
        "/var/log",        //Linux 下建议的日志目录
        "/var/tmp",        //Linux 下建议的临时文件目录
        RELAYER_DEFAULT_THREAD_MAX,
        true,
        null);            //不要打印回调
#endif //WIN32
    g_bMainExitFlag.Set(false);    //主退出标志置为假
    //...其他初始化代码

```

下面是主循环，和前文例子类似，这里大家请关注，Windows 下是通过键盘输入 exit 退出，而 Linux 下，则是死循环，等待中断程序修改 g_bMainExitFlag 的值为 true，退出。

```

    while(!g_bMainExitFlag.Get())    //Linux 下判断退出条件
    {
#ifdef WIN32                                //Windows 下读取键盘，exit 退出
        SafeScanf(szBuffer,256);
        if(0==strcmp("exit",szBuffer))
        {
            break;
        }
        else
            TONY_XIAO_PRINTF("\"%s\" is not command!\n",szBuffer);
#else // not WIN32
#endif //WIN32
        Sleep(50);
    }

```

退出代码，清空现场，清除聚合工具类。

```

        //...其他退出代码
        if(m_pBaseLibrary)
        {
            delete m_pBaseLibrary;
            m_pBaseLibrary=null;
        }
    }
    DeleteMyPid();                //这步最重要，清除本次运行的 pid。
    return 0;
}

```

11.3.5.7 远端实例退出模式

远端实例退出模式也还是复杂，中间主要涉及到检测是否有一个远端实例正在普通运行模式运行。基本调用程序如下

```

int main_exit_remote(int argc, char* argv[])
{
    RunRemoteExit();
    return 0;
}

```

这里调用了—个功能函数 RunRemoteExit。

```

void RunRemoteExit()
{
    char szBuffer[256];           //准备缓冲区
    int nRemotePid=-1;
    nRemotePid=GetRemotePid();    //获得远端进程实例的 pid
    if(0>nRemotePid)              //如果得到-1，就表示没有远端实例，打印提示信息，退出
    {
        TONY_XIAO_PRINTF("Remote mi_dbd is not run!\n");
        return;
    }
    //调用 kill 系统功能调用，发出 SIGTERM 信号，通知远端实例退出
    //如果远端实例存在，会激活远端实例的 _RemoteExitFunc 函数
    //然后该函数设置退出标志，main 循环退出。
    SafePrintf(szBuffer,256,"kill -s %d %d\n",SIGTERM,nRemotePid);
    system(szBuffer);
    //这个等待很重要，因为远端实例可能会退出很久，需要等待。
    //本运行模式通常会在一个 Linux 退出脚本中自动运行，如果发出消息，本实例马上退出
    //则很可能远端实例的数据尚未保存完毕，系统就关闭了，仍然丢失数据。
    //此处的等待循环，主要是等待远端实例真实退出，在此期间，将本脚本流程“悬挂”住
    //避免上述 bug。这是典型的异步转同步的等待逻辑。
    while(IHaveRunning())
    {
        Sleep(1000);
    }
    TONY_XIAO_PRINTF("Remote mi_dbd has been exited!\n"); //打印退出信息
}

```

11.3.5.8 清空 pid 模式

清空 pid 模式非常简单，就是调用 DeleteMyPid 即可。

```
int main_clean_run_flag(int argc, char* argv[])
{
    DeleteMyPid();
    return 0;
}
```

11.3.6 程序使用说明

经过上述设计，Linux 下的服务器已经基本能满足运营使用需求，一般的 Linux 的管理员，已经可以通过启动脚本和退出脚本，实现服务器的温和退出。由于 Linux 脚本开发不是本书研究的重点内容，此处不再赘述，有兴趣的读者，可以看一些 Linux shell 脚本开发书籍，来补充这部分知识。

另外，很多时候，企业服务器托管在电信机房，管理员只能远程登录管理。这在 Windows 下比较困难，但 Linux 下非常简单，管理员可以通过 putty 等 ssh 控制台终端，远程登录到服务器上操作，此时，可以使用 -q 参数临时退出我们的服务程序实现维护。

第 12 章 细节决定成败（代结束语）

12.1 工程实践注重细节

12.2 究竟怎样才能学好 C 和 C++ 语言开发

12.3 如何做一个成功的软件工程师

12.4 关于网络数据传输

12.4 结束语

第 12 章 细节决定成败（代结束语）

细心的读者可能感觉到，本书很多的内容其实都是一些细节，如果按照传统计算机教学来说，似乎没必要这么讲这么细。不过，笔者请大家关注，恰恰是这些细节，往往决定了一个商用工程的成败。

笔者看过很多刚出校门的同学的代码，感觉很怪，代码都对，但只要一代入到工程中，总是有这样那样的问题，给人的感觉，总像“玩具”，不像“工具”。

因此，笔者在此增补了这一章，希望大家能从本章的介绍中，学习到一点程序之外的东西。

12.1 工程实践注重细节

很多年前，笔者很年轻的时候，和一些老工程师沟通，总是有一种奇怪的感觉，感觉这些人特别注重细节。很多时候，笔者向他们讲述一个设计过程，大家知道，人说话总是有重点，有省略，但是，每次笔者认为理所当然的地方，准备略过不提的时候，总是被对方发问，这些老同事给笔者的感觉，总是揪住笔者的“软肋”在打，问得笔者张口结舌。甚至，很多时候，问得笔者都没有信心了，感觉自己学的东西，在工程中一点都不正确。

这种感觉，相信很多初入职场的同学，甚至已经在职场中走过几年的年轻朋友，都深有体会。这是一种很无奈的感觉。

但是，随着笔者这么多年的工作，经验不断积累，教训不断加深，才发现这些老工程师问得很有道理，他们问的，恰恰是基于学校里面的学生思维，最容易忽视的细节，而这些细节，缺往往是工程成败的关键。

比如一个简单的传输工程，学生会认为，我们连接一个 Socket，发送一个请求，并且收到了回应，一笔交易完成，根据“复制”的原理，只要有一笔交易完成，那么所有的交易都应该顺利通过。事实上，学校里面大多数试验，我们都是这么做的，一般做到这个程度，老师已经能给满分了。

但是，在一个资深的传输工程师看来，这一切漏洞太多，无法实用。

1、一笔交易完成，这是在局域网中的实验还是公网试验？有没有出现过错误？错误率是多少？

2、出错之后的重试机制是怎么实现的？是通过传输层的“等停”来实现确定传输，还是通过系统协议的设计，允许信令错误，通过循环动作来自动弥补某一次的传输失败？

3、确定传输的信令和逻辑是怎样的？失败之后的策略如何？socket 是否保留？如果不保留，如何重建？

4、这个程序是单线程的还是多线程的？如果是单线程，那么很可能因为某一次传输的失败，导致整体业务的悬挂，那么，如何恢复？

5、如果是多线程的，那么，多线程的锁安全性如何？动态负载均衡如何处理？会不会有资源泄漏？会不会有死线程？

6、服务器最大能应对多少路并发连接？这些并发连接折合最大在线人数是多少？这些最大在线任务折合系统总容量是多少？

7、根据系统总容量设计，系统准备使用什么规模的数据库？数据库选型如何确定？

8、如果以后系统扩展，单一数据库容量不够，那么系统设计上，如何考虑以后的平行扩容？

9、...

笔者写不下去了，问题太多，当笔者有一天，终于理解到当初盘问自己的老工程师究竟在想什么时，笔者才发现，原来工程项目，并没有什么高深的知识，更多的则是对细节的关注，**细节决定成败**。

12.2 究竟怎样才能学好 C 和 C++语言开发

本书定位于一本讲解使用 C 和 C++语言进行商用工程开发的书籍，但是细心的读者可能注意到，本书花了大量的篇幅，在给大家讲解细节，程序设计的细节，无错化设计的细节，并行计算的细节，内存池、时间片的细节，等等.....

很多时候，一个初学者学习 C 和 C++语言，会有很多茫然的感觉，书上讲的东西都很简单，哪怕是指针，类，模板，几天就学习完了，但是，一到实际开发，就发现自己提笔忘字，心里一片茫然，不知道写什么？这种感觉，笔者有，相信各位读者也会有。

在网上，很多同学在请教笔者时，也说出了这种茫然，总是觉得，自己已经学习了很多 C 和 C++知识，其他的算法语言、数据结构、编译原理、操作系统，甚至计算机组成原理，都学得很好，但是，就是用不到工程实战中去。请教笔者这是为什么？

在笔者看来，这些知识都很重要，非常重要，但是，学校教育和企业中工程开发，还是各有侧重的。学校里面的教育，通常是培养一个学生对于知识深度的掌握，学习 C 语言，就要把这门语言的方方面面都学到，因为以后备不住就用得到，学习 C++语言，也差不多，学任何知识，都差不多。

但是，有一门知识，是学校里面永远无法教育的，就是各门知识之间的关联性，简单说，就是熟练使用知识的能力。我们在某一天，遇到某一个需求，我们能迅速从自己的知识体系中筛选出来需要的知识，迅速“拼接”、“搭建”出一个合理的工程模型，并能将之实现出来，这本身，就是一门知识。

但显然大学中不会有这门课，我们需要自己来学习。那么，如何学习呢？

笔者的理解，这种对知识的熟练运用的能力，其实是基于对各门知识熟练之极，高度抽象化理解之后，能提取出知识的“共性”，并能理解其“个性”差异的能力。比如笔者前文所述的内存池，我们可以用队列，也可以用栈实现，甚至，我们简单做个数组都能实现，但，如何筛选？为什么这么筛选？道理在哪里？

其实大家可以看出，当知识储备差不多的时候，我们研究得更多的是细节。

本书的写作，笔者一直试图培养大家这种“使用知识”的能力，这中间当然有很多经验，很多教训，写程序时，有很多顾虑，有很多忌讳，种种这些，其实都是在讲解一个道理，**细节决定成败**。

12.3 如何做一个成功的软件工程师

在笔者看来，一个计算机专业的学生，和一个成熟的软件工程师，最大的差异，就是对于很多细节的把握，以及思路的广度和深度。

对于一门语言，一个操作系统，一个平台，学生可能仅仅局限在“会用”，而软件工程师更多的要求自己“掌握”。

什么叫掌握呢？这个掌握不仅仅是写出一两段代码，能正常运行，而是通过大量的实验，长期的实战，对一个系统提供的 `api`，每一个功能函数具体的特性的一种深入掌握，它是怎样被调用的？它的调用有什么特性？什么情况下它会崩溃？它是不是多线程安全的？等等……

这些细节不是简简单单阅读两天 MSDN，或者 `man` 就能掌握的，更多的是来自于实践中的经验和教训，当一个软件工程师，能掌握他工作中大多数应用场合需要的知识点，他其实已经很成功了。

这体现在，他下次设计时，会有很多顾虑，我们经常说，写程序“越写越胆小”，“越写越保守”，就是这个道理。他会考虑很多问题，大多数这种问题，在很多教科书上是一笔带过，甚至提都没有提，但是，他在实践中，知道这是重点，是关键。

当他小心翼翼地做完系统设计，开始实施，他也不会马上动笔，而是仔细推演自己的设计，有哪些异常情况还没有考虑到，并且，尽可能把程序一次做正确，因为，“改 `bug` 是改不出精品的”。

因此，资深的工程师，一般都有自己的代码规范，这是他自己对自己的约定，根据经验，这种规矩，能帮助他少犯错误。因为他知道，在开发中，自己 90% 以上的精力都会集中在功能的实现和性能的保证上，很少有时机和机会精雕细琢自己的代码，因此，他要求自己的代码一次正确。

笔者的“C 和 C++ 无错化程序设计方法”，就是这样产生的，但笔者相信，很多资深的软件工程师，都有自己的无错化设计方法，仅仅是没有说出来而已。

当工程完成，软件工程师会更加小心地看待自己的代码，使用大量的测试代码（通常超过原代码本身的长度）来仔细检测自己的工程代码，通过大量正确的验证结果，来建立自己对代码的信心，这才能勉强走到 `RD release` 的地步。

然后是测试部门的测试，当一个工程项目，经过了这一系列精品的设计、实施、测试，也只能勉强算作 `Beta` 测试，开始发布到公网试运行。

当试运行达到一年以上，没有大的 `bug` 报出，没有运营部门的性能抱怨，这才算做一个项目可以结束。

大家可以看到，在这个过程中，真正的程序书写工作，其实很少，我们更多的精力，都是放在业务、性能、运营上，才能勉强得到一个比较好的结果。

中国人做事情，很多时候，很“两难”，其实商用工程的设计和实施，也是一个“两难”的过程，多数时候，我们面临的问题，是“这也不行，那也不行”，软件工程师更多地是在走钢丝，在多种制约条件下，实现一个大多数情况下可用的工程结果。这个大多数情况，一般不超过 80%。

因此，请大家注意，商用数据传输工程，甚至所有的商用工程，往往没有绝对的“对”与“错”，这不是一个二进制的世界，世界上也不仅仅是黑和白两种情况，更多的时候，我们的工程输出，是一个灰色的地带，是一个百分比。我们能做的，仅仅是把这个百分比提高一点而已。

通常情况下，当我们对细节的关注度加大，则我们的百分比提升，否则就降低。学生通常也能写出工程代码，但其成功服务的百分比往往很低，其根源，就是对细节的把握程度。

笔者在本书的前半部分，重点讲解了很多与数据传输无关的基础知识，主要就是希望大家传递一些细节上的知识，因为这些细节，将会极大地影响下面我们要讲解的网络通信编程的成功度。

请各位读者仔细体会这一点。

12.4 关于网络数据传输

细心的读者可能看出，本书虽然站在商用工程开发的角度，特别是以数据传输为主的商用工程角度，对 C 和 C++ 工程实战的一些经验，原则，方法做了论述，但是，对于数据传输最主要的功能：基于 TCP/IP 协议实现的 socket 通信，却没有涉及。

这里笔者要解释一下，socket 通信看似简单，但其实非常复杂。一个程序员，随便找本教材，三天，甚至三个小时，就可能写出两段可以互相通信的程序。但是，如果要做出一个可以商用化，可以在公网实现 7*24 小时长时间运营的产品，恐怕三年都不够。

为什么呢？笔者认为其根源就在于商用数据传输和网络通信有所不同，实现一个商用服务器产品，需要的知识太多太杂，产品的要求也太高，其中需要程序员储备很多通信之外的知识，本书中的很多知识，都可以视为这种知识储备的补充。

同时，在商用数据传输领域，就笔者的经验，程序的正确性，通常已经不是一个需要讨论的问题，程序员必须首先保证程序正确，然后还必须讨论架构的正确，在数据传输领域，基于宏观层面、系统层面的架构设计，更重要一些。

但这部分知识，本身就是很庞大的一个整体，而本书由于篇幅所限，主要定位于讲解 C 和 C++ 的商用并行工程开发技巧，因此，涉及这部分内容就比较少一点，请各位读者谅解。

在笔者计划中的另外一本书中，准备向大家详细介绍数据传输相关的知识，敬请各位读者期待。

12.4 结束语

本书是从笔者早年，在西南交大软件学院讲授的一门课程《C/C++ 无错化程序设计》演化过来的。由于笔者本人水平有限，成书时间也偏短，书中错漏肯定不少，各位读者如果在阅读过程中，发现问题，还请不吝赐教，多多批评。

笔者联系方式：(略)