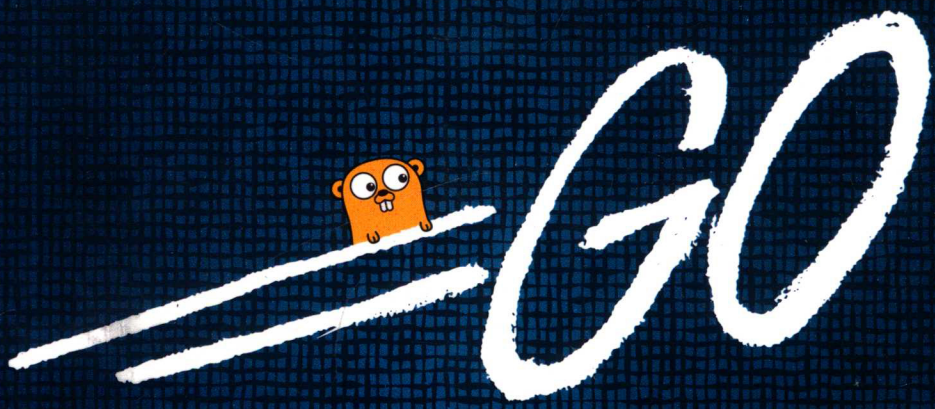
TURING 图灵原创



Go语言编程

The Go Programming Language

许式伟 吕桂华◉等编著

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵原创

Go语言编程

The Go Programming Language

许式伟 吕桂华◉等编著

人民邮电出版社
北 京

图书在版编目（CIP）数据

Go语言编程 / 许式伟等编著. -- 北京 : 人民邮电出版社, 2012.9
（图灵原创）
ISBN 978-7-115-29036-6

Ⅰ. ①G… Ⅱ. ①许… Ⅲ. ①程序语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2012)第172819号

内 容 提 要

本书首先引领读者快速浏览 Go 语言的全貌，迅速消除读者对这门语言的陌生感，然后循序渐进地介绍了 Go 语言的面向过程和面向对象的编程语法，其中穿插了一些与其他主流语言的比较以让读者理解 Go 语言的设计动机，接着探讨了 Go 语言最为重要的并行编程方法，之后介绍了网络编程、工程管理、安全编程、开发工具等非语法相关但非常重要的内容，最后为一系列关于 Go 语言的文章，可以帮助读者更深入了解这门全新的语言。

本书适合所有层次的开发者阅读。

图灵原创

Go语言编程

◆ 编著　　许式伟　吕桂华　等
责任编辑　王军花

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京天宇星印刷厂印刷

◆ 开本：800×1000　1/16
印张：15.25
字数：361千字　　2012年9月第1版
印数：1－4 000册　　2012年9月北京第1次印刷

ISBN 978-7-115-29036-6

定价：49.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223
反盗版热线：(010)67171154

前言：为什么我们需要一门新语言

编程语言已经非常多，偏性能敏感的编译型语言有C、C++、Java、C#、Delphi和Objective-C等，偏快速业务开发的动态解析型语言有PHP、Python、Perl、Ruby、JavaScript和Lua等，面向特定领域的语言有Erlang、R和MATLAB等，那么我们为什么需要Go这样一门新语言呢？

在2000年前的单机时代，C语言是编程之王。随着机器性能的提升、软件规模与复杂度的提高，Java逐步取代了C的位置。尽管看起来Java已经深获人心，但Java编程的体验并未尽如人意。历年来的编程语言排行榜（如图0-1所示）显示，Java语言的市场份额在逐步下跌，并趋近于C语言的水平，显示了这门语言后劲不足。

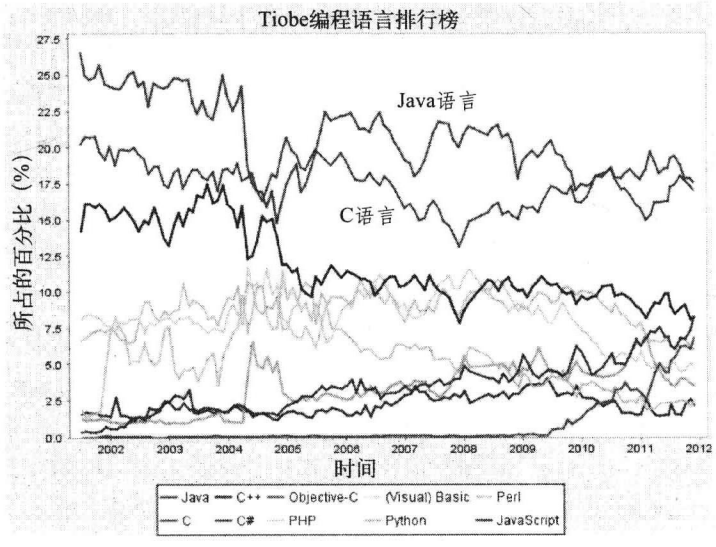


图0-1　编程语言排行榜[①]

Go语言官方自称，之所以开发Go语言，是因为“近10年来开发程序之难让我们有点沮丧”。这一定位暗示了Go语言希望取代C和Java的地位，成为最流行的通用开发语言。

Go希望成为互联网时代的C语言。多数系统级语言（包括Java和C#）的根本编程哲学来源于

① 数据来源：http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html。

C++，将C++的面向对象进一步发扬光大。但是Go语言的设计者却有不同的看法，他们认为C++真的没啥好学的，值得学习的是C语言。C语言经久不衰的根源是它足够简单。因此，Go语言也要足够简单！

那么，互联网时代的C语言需要考虑哪些关键问题呢？

首先，并行与分布式支持。多核化和集群化是互联网时代的典型特征。作为一个互联网时代的C语言，必须要让这门语言操作多核计算机与计算机集群如同操作单机一样容易。

其次，软件工程支持。工程规模不断扩大是产业发展的必然趋势。单机时代语言可以只关心问题本身的解决，而互联网时代的C语言还需要考虑软件品质保障和团队协作相关的话题。

最后，编程哲学的重塑。计算机软件经历了数十年的发展，形成了面向对象等多种学术流派。什么才是最佳的编程实践？作为互联网时代的C语言，需要回答这个问题。

接下来我们来聊聊Go语言在这些话题上是如何应对的。

并发与分布式

多核化和集群化是互联网时代的典型特征，那语言需要哪些特性来应对这些特征呢？

第一个话题是并发执行的“执行体”。执行体是个抽象的概念，在操作系统层面有多个概念与之对应，比如操作系统自己掌管的进程（process）、进程内的线程（thread）以及进程内的协程（coroutine，也叫轻量级线程）。多数语言在语法层面并不直接支持协程，而通过库的方式支持的协程的功能也并不完整，比如仅仅提供协程的创建、销毁与切换等能力。如果在这样的协程中调用一个同步IO操作，比如网络通信、本地文件读写，都会阻塞其他的并发执行协程，从而无法真正达到协程本身期望达到的目标。

Go语言在语言级别支持协程，叫goroutine。Go语言标准库提供的所有系统调用（syscall）操作，当然也包括所有同步IO操作，都会出让CPU给其他goroutine，这让事情变得非常简单。我们对比一下Java和Go，近距离观摩下两者对“执行体”的支持。

为了简化，我们在样例中使用的是Java标准库中的线程，而不是协程，具体代码如下：

```
public class MyThread implements Runnable {


    String arg;

    public MyThread(String a) {
        arg = a;
    }

    public void run() {
        // ...
    }

    public static void main(String[] args) {
        new Thread(new MyThread("test")).start();
        // ...
    }
```

```
}
```

相同功能的代码，在Go语言中是这样的：

```
func run(arg string) {
    // ...
}

func main() {
    go run("test")
    ...
}
```

对比非常鲜明。我相信你已经明白为什么Go语言会叫Go语言了：Go语言献给这个时代最好的礼物，就是加了go这个关键字。当然也有人会说，叫Go语言是因为它是Google出的。好吧，这也是个不错的闲聊主题。

第二个话题是“执行体间的通信”。执行体间的通信包含几个方式：

- 执行体之间的互斥与同步
- 执行体之间的消息传递

先说“执行体之间的互斥与同步”。当执行体之间存在共享资源（一般是共享内存）时，为保证内存访问逻辑的确定性，需要对访问该共享资源的相关执行体进行互斥。当多个执行体之间的逻辑存在时序上的依赖时，也往往需要在执行体之间进行同步。互斥与同步是执行体间最基础的交互方式。

多数语言在库层面提供了线程间的互斥与同步支持，那么协程之间的互斥与同步呢？呃，不好意思，没有。事实上多数语言标准库中连协程都是看不到的。

再说“执行体之间的消息传递”。在并发编程模型的选择上，有两个流派，一个是共享内存模型，一个是消息传递模型。多数传统语言选择了前者，少数语言选择后者，其中选择“消息传递模型”的最典型代表是Erlang语言。业界有专门的术语叫“Erlang风格的并发模型”，其主体思想是两点：一是“轻量级的进程（Erlang中‘进程’这个术语就是我们上面说的‘执行体’）”，二是“消息乃进程间通信的唯一方式”。当执行体之间需要相互传递消息时，通常需要基于一个消息队列（message queue）或者进程邮箱（process mail box）这样的设施进行通信。

Go语言推荐采用“Erlang风格的并发模型”的编程范式，尽管传统的“共享内存模型”仍然被保留，允许适度地使用。在Go语言中内置了消息队列的支持，只不过它叫通道（channel）。两个goroutine之间可以通过通道来进行交互。

软件工程

单机时代的语言可以只关心问题本身的解决，但是随着工程规模的不断扩大，软件复杂度的不断增加，软件工程也成为语言设计层面要考虑的重要课题。多数软件需要一个团队共同去完成，在团队协作的过程中，人们需要建立统一的交互语言来降低沟通的成本。规范化体现在多个层面，如：

- 代码风格规范
- 错误处理规范
- 包管理
- 契约规范（接口）
- 单元测试规范
- 功能开发的流程规范

Go语言很可能是第一个将代码风格强制统一的语言，例如Go语言要求public的变量必须以大写字母开头，private变量则以小写字母开头，这种做法不仅免除了public、private关键字，更重要的是统一了命名风格。

另外，Go语言对{ }应该怎么写进行了强制，比如以下风格是正确的：

```
if expression {
    ...
}
```

但下面这个写法就是错误的：

```
if expression
{
    ...
}
```

而C和Java语言中则对花括号的位置没有任何要求。哪种更有利，这个见仁见智。但很显然的是，所有的Go代码的花括号位置肯定是非常统一的。

最有意思的其实还是 Go 语言首创的错误处理规范：

```
f, err := os.Open(filename)
if err != nil {
    log.Println("Open file failed:", err)
    return
}
defer f.Close()
... // 操作已经打开的f文件
```

这里有两个关键点。其一是defer关键字。defer语句的含义是不管程序是否出现异常，均在函数退出时自动执行相关代码。在上面的例子中，正是因为有了defer，才使得无论后续是否会出现异常，都可以确保文件被正确关闭。其二是Go语言的函数允许返回多个值。大多数函数的最后一个返回值会为error类型，以在错误情况下返回详细信息。error类型只是一个系统内置的interface，如下：

```
type error interface {
    Error() string
}
```

有了error类型，程序出现错误的逻辑看起来就相当统一。

在Java中，你可能这样写代码来保证资源正确释放：

```
Connection conn = ...;
```

```
try {
    Statement stmt = ...;
    try {
        ResultSet rset = ...;
        try {
            ... // 正常代码
        }
        finally {
            rset.close();
        }
    }
    finally {
        stmt.close();
    }
}
finally {
    conn.close();
}
```

完成同样的功能，相应的Go代码只需要写成这样：

```
conn := ...
defer conn.Close()

stmt := ...
defer stmt.Close()

rset := ...
defer rset.Close()
... // 正常代码
```

对比两段代码，Go语言处理错误的优势显而易见。当然，其实Go语言带给我们的惊喜还有很多，后续有机会我们可以就某个更具体的话题详细展开来谈一谈。

编程哲学

计算机软件经历了数十年的发展，形成了多种学术流派，有面向过程编程、面向对象编程、函数式编程、面向消息编程等，这些思想究竟孰优孰劣，众说纷纭。

C语言是纯过程式的，这和它产生的历史背景有关。Java语言则是激进的面向对象主义推崇者，典型表现是它不能容忍体系里存在孤立的函数。而Go语言没有去否认任何一方，而是用批判吸收的眼光，将所有编程思想做了一次梳理，融合众家之长，但时刻警惕特性复杂化，极力维持语言特性的简洁，力求小而精。

从编程范式的角度来说，Go语言是变革派，而不是改良派。

对于C++、Java和C#等语言为代表的面向对象（OO）思想体系，Go语言总体来说持保守态度，有限吸收。

首先，Go语言反对函数和操作符重载（overload），而C++、Java和C#都允许出现同名函数或操作符，只要它们的参数列表不同。虽然重载解决了一小部分面向对象编程（OOP）的问题，但

同样给这些语言带来了极大的负担。而Go语言有着完全不同的设计哲学，既然函数重载带来了负担，并且这个特性并不对解决任何问题有显著的价值，那么Go就不提供它。

其次，Go语言支持类、类成员方法、类的组合，但反对继承，反对虚函数（virtual function）和虚函数重载。确切地说，Go也提供了继承，只不过是采用了组合的文法来提供：

```
type Foo struct {
    Base
    ...
}

func (foo *Foo) Bar() {
    ...
}
```

再次，Go语言也放弃了构造函数（constructor）和析构函数（destructor）。由于Go语言中没有虚函数，也就没有vptr，支持构造函数和析构函数就没有太大的价值。本着“如果一个特性并不对解决任何问题有显著的价值，那么Go就不提供它”的原则，构造函数和析构函数就这样被Go语言的作者们干掉了。

在放弃了大量的OOP特性后，Go语言送上了一份非常棒的礼物：接口（interface）。你可能会说，除了C这么原始的语言外，还有什么语言没有接口呢？是的，多数语言都提供接口，但它们的接口都不同于Go语言的接口。

Go语言中的接口与其他语言最大的一点区别是它的非侵入性。在C++、Java和C#中，为了实现一个接口，你需要从该接口继承，具体代码如下：

```
class Foo implements IFoo { // Java文法
    ...
}

class Foo : public IFoo { // C++文法
    ...
}

IFoo* foo = new Foo;
```

在Go语言中，实现类的时候无需从接口派生，具体代码如下：

```
type Foo struct { // Go文法
    ...
}

var foo IFoo = new(Foo)
```

只要Foo实现了接口IFoo要求的所有方法，就实现了该接口，可以进行赋值。

Go语言的非侵入式接口，看似只是做了很小的文法调整，实则影响深远。

其一，Go语言的标准库再也不需要绘制类库的继承树图。你只需要知道这个类实现了哪些方法，每个方法是啥含义就足够了。

其二，不用再纠结接口需要拆得多细才合理，比如我们实现了File类，它有下面这些方法：

```
Read(buf []byte) (n int, err error)
Write(buf []byte) (n int, err error)
Seek(off int64, whence int) (pos int64, err error)
Close() error
```

那么，到底是应该定义一个IFile接口，还是应该定义一系列的IReader、IWriter、ISeeker和ICloser接口，然后让File从它们派生好呢？事实上，脱离了实际的用户场景，讨论这两个设计哪个更好并无意义。问题在于，实现File类的时候，我怎么知道外部会如何用它呢？

其三，不用为了实现一个接口而专门导入一个包，而目的仅仅是引用其中的某个接口的定义。在Go语言中，只要两个接口拥有相同的方法列表，那么它们就是等同的，可以相互赋值，如对于以下两个接口，第一个接口：

```
package one

type ReadWriter interface {
    Read(buf [] byte) (n int, err error)
    Write(buf [] byte) (n int, err error)
}
```

第二个接口：

```
package two

type IStream interface {
    Write(buf [] byte) (n int, err error)
    Read(buf [] byte) (n int, err error)
}
```

这里我们定义了两个接口，一个叫one.ReadWriter，一个叫two.IStream，两者都定义了Read()和Write()方法，只是定义的次序相反。one.ReadWriter先定义了Read()再定义Write()，而two.IStream反之。

在Go语言中，这两个接口实际上并无区别，因为：

- 任何实现了one.ReadWriter接口的类，均实现了two.IStream；
- 任何one.ReadWriter接口对象可赋值给two.IStream，反之亦然；
- 在任何地方使用one.ReadWriter接口，与使用two.IStream并无差异。

所以在Go语言中，为了引用另一个包中的接口而导入这个包的做法是不被推荐的。因为多引用一个外部的包，就意味着更多的耦合。

除了OOP外，近年出现了一些小众的编程哲学，Go语言对这些思想亦有所吸收。例如，Go语言接受了函数式编程的一些想法，支持匿名函数与闭包。再如，Go语言接受了以Erlang语言为代表的面向消息编程思想，支持goroutine和通道，并推荐使用消息而不是共享内存来进行并发编程。总体来说，Go语言是一个非常现代化的语言，精小但非常强大。

小结

在十余年的技术生涯中，我接触过、使用过、喜爱过不同的编程语言，但总体而言，Go语言的出现是最让我兴奋的事情。我个人对未来10年编程语言排行榜的趋势判断如下：

- Java语言的份额继续下滑，并最终被C和Go语言超越；
- C语言将长居编程榜第二的位置，并有望在Go取代Java前重获语言榜第一的宝座；
- Go语言最终会取代Java，居于编程榜之首。

由七牛云存储团队编著的这本书将尽可能展现出Go语言的迷人魅力。希望本书能够让更多人理解这门语言，热爱这门语言，让这门优秀的语言能够落到实处，把程序员从以往繁杂的语言细节中解放出来，集中精力开发更加优秀的系统软件。

许式伟

2012年3月7日

目　录

第1章

初识Go语言

本章将简要介绍Go语言的发展历史和关键的语言特性，并引领读者对Go语言的主要特性进行一次快速全面的浏览，让读者对Go语言的总体情况有一个清晰的印象，并能够快速上手，用Go语言编写和运行自己的第一个小程序。

1.1 语言简史

提起Go语言的出身，我们就必须将我们饱含敬意的眼光投向持续推出惊世骇俗成果的贝尔实验室。贝尔实验室已经走出了多位诺贝尔奖获得者，一些对于现在科技至关重要的研究成果，比如晶体管、通信技术、数码相机的感光元件CCD和光电池等都源自贝尔实验室。该实验室在科技界的地位可想而之，是一个毫无争议的科研圣地。

这里我们重点介绍一下贝尔实验室中一个叫计算科学研究中心的部门对于操作系统和编程语言的贡献。回溯至1969年（估计大部分读者那时候都还没出世），肯·汤普逊（Ken Thompson）和丹尼斯·里奇（Dennis Ritchie）在贝尔实验室的计算科学研究中心里开发出了Unix这个大名鼎鼎的操作系统，还因为开发Unix而衍生出了一门同样赫赫有名的编程语言——C语言。对于很大一部分人而言，Unix就是操作系统的鼻祖，C语言也是计算机课程中最广泛使用的编程语言。Unix和C语言在过去的几十年以来已经造就了无数的成功商业故事，比如曾在90年代如日中天的太阳微系统（Sun MicroSystems），现在正如日中天的苹果的Mac OS X操作系统其实也可以认为是Unix的一个变种（FreeBSD）。

虽然已经取得了如此巨大的成就，贝尔实验室的这几个人并没有因此而沉浸在光环中止步不前，他们从20世纪80年代又开始了一个名为Plan 9的操作系统研究项目，目的就是解决Unix中的一些问题，发展出一个Unix的后续替代系统。在之后的几十年中，该研究项目又演变出了另一个叫Inferno的项目分支，以及一个名为Limbo的编程语言。

Limbo是用于开发运行在小型计算机上的分布式应用的编程语言，它支持模块化编程，编译期和运行时的强类型检查，进程内基于具有类型的通信通道，原子性垃圾收集和简单的抽象数据类型。它被设计为：即便是在没有硬件内存保护的小型设备上，也能安全运行。

Limbo语言被认为是Go语言的前身，不仅仅因为是同一批人设计的语言，而是Go语言确实从Limbo语言中继承了众多优秀的特性。

贝尔实验室后来经历了多次的动荡，包括肯·汤普逊在内的Plan 9项目原班人马加入了

Google。在Google，他们创造了Go语言。早在2007年9月，Go语言还是这帮大牛的20%自由时间的实验项目。幸运的是，到了2008年5月，Google发现了Go语言的巨大潜力，从而开始全力支持这个项目，让这批人可以全身心投入Go语言的设计和开发工作中。Go语言的第一个版本在2009年11月正式对外发布，并在此后的两年内快速迭代，发展迅猛。第一个正式版本的Go语言于2012年3月28日正式发布，让Go语言迎来了第一个引人瞩目的里程碑。

基于Google对开源的一贯拥抱态度，Go语言也自然而然地选择了开源方式发布，并使用BSD授权协议。任何人可以查看Go语言的所有源代码，并可以为Go语言发展而奉献自己的力量。

Google作为Go语言的主推者，并没有简简单单地把语言推给开源社区了事，它不仅组建了一个独立的小组全职开发Go语言，还在自家的服务中逐步增加对Go语言的支持，比如对于Google有战略意义的云计算平台GAE（Google AppEngine）很早就开始支持Go语言了。按目前的发展态势，在Google内部，Go语言有逐渐取代Java和Python主流地位的趋势。在Google的更多产品中，我们将看到Go语言的踪影，比如Google最核心的搜索和广告业务。

在本书的序中，我们已经清晰诠释了为什么在语言泛滥的时代Google还要设计和推出一门新的编程语言。按照已经发布的Go语言的特性，我们有足够的理由相信Google推出此门新编程语言绝不仅仅是简单的跑马圈地运动，而是为了解决切实的问题。

下面我们再来看看Go语言的主要作者。

- 肯·汤普逊（Ken Thompson，http://en.wikipedia.org/wiki/Ken_Thompson）：设计了B语言和C语言，创建了Unix和Plan 9操作系统，1983年图灵奖得主，Go语言的共同作者。
- 罗布·派克（Rob Pike，http://en.wikipedia.org/wiki/Rob_Pike）：Unix小组的成员，参与Plan 9和Inferno操作系统，参与Limbo和Go语言的研发，《Unix编程环境》作者之一。
- 罗伯特·格里泽默（Robert Griesemer）：曾协助制作Java的HotSpot编译器和Chrome浏览器的JavaScript引擎V8。
- 拉斯·考克斯（Russ Cox，http://swtch.com/~rsc/）：参与Plan 9操作系统的开发，Google Code Search项目负责人。
- 伊安·泰勒（Ian Lance Taylor）：GCC社区的活跃人物，gold连接器和GCC过程间优化LTO的主要设计者，Zembu公司的创始人。
- 布拉德·菲茨帕特里克（Brad Fitzpatrick，http://en.wikipedia.org/wiki/Brad_Fitzpatrick）：LiveJournal的创始人，著名开源项目memcached的作者。

虽然我们这里只列出了一部分，大家已经可以看出这个语言开发团队空前强大，这让我们在为Go语言的优秀特性而兴奋之外，还非常看好这门语言的发展前景。

1.2　语言特性

Go语言作为一门全新的静态类型开发语言，与当前的开发语言相比具备众多令人兴奋不已的新特性。本书从第2章开始，我们将对Go语言的各个方面进行详细解析，让读者能够尽量轻松地掌握这门简洁、有趣却又超级强大的新语言。

这里先给读者罗列一下Go语言最主要的特性：

- 自动垃圾回收
- 更丰富的内置类型
- 函数多返回值
- 错误处理
- 匿名函数和闭包
- 类型和接口
- 并发编程
- 反射
- 语言交互性

1.2.1 自动垃圾回收

我们可以先看下不支持垃圾回收的语言的资源管理方式，以下为一小段C语言代码：

```
void foo()
{
    char* p = new char[128];
    ... // 对p指向的内存块进行赋值
    func1(p); // 使用内存指针

    delete[] p;
}
```

各种非预期的原因，比如由于开发者的疏忽导致最后的delete语句没有被调用，都会引发经典而恼人的内存泄露问题。假如该函数被调用得非常频繁，那么我们观察该进程执行时，会发现该进程所占用的内存会一直疯长，直至占用所有系统内存并导致程序崩溃，而如果泄露的是系统资源的话，那么后果还会更加严重，最终很有可能导致系统崩溃。

手动管理内存的另外一个问题就是由于指针的到处传递而无法确定何时可以释放该指针所指向的内存块。假如代码中某个位置释放了内存，而另一些地方还在使用指向这块内存的指针，那么这些指针就变成了所谓的“野指针”（wild pointer）或者“悬空指针”（dangling pointer），对这些指针进行的任何读写操作都会导致不可预料的后果。

由于其杰出的效率，C和C++语言在非常长的时间内都作为服务端系统的主要开发语言，比如Apache、Nginx和MySQL等著名的服务器端软件就是用C和C++开发的。然而，内存和资源管理一直是一个让人非常抓狂的难题。服务器的崩溃十有八九就是因为不正确的内存和资源管理导致，更讨厌的是这种内存和资源管理问题即使被发现了，也很难定位到具体的错误地点，导致无数程序员通宵达旦地调试程序。

这个问题在多年里被不同人用不同的方式来试图解决，并诞生了一些非常著名的内存检查工具，比如Rational Purify、Compuware BoundsChecker和英特尔的Parallel Inspector等。从设计方法的角度也衍生了类似于内存引用计数之类的方法（通常被称为“智能指针”），后续在Windows平台上标准化的COM出现的一个重要原因就是为了解决内存管理的难题。但是事实证明，这些工具和