码农 the code maker

27

为什么需要一门新语言

Go 的线程实现模型

助力谷歌 App Engine 云平台

白宸: Redis 带你尽享丝滑

用正确的方式，打开 Go 测试

解决并发之痛

程序员到底能不能干过 30 岁

目　录

编者的话

专题：Just Go！

1　为什么需要一门新语言

14　Go的线程实现模型

29　助力谷歌App Engine云平台

人物

46　白宸：Redis带你尽享丝滑！

践行

57　用正确的方式，打开Go测试

70　解决并发之痛

鲜阅

87　程序员到底能不能干过30岁？

书单

95　偷偷看下你的书单

98　电子书单

编者的话

编者 / 刘敏

与其站在选择的路口踟蹰，不如大胆地 Just Go!

编程世界的语言千千万万，有执行速度快但编译速度不太理想的，有编译速度迅速但执行速度不佳的，也有执行速度一般但开发难度较低的。显然，能够在编译、执行、开发几个条件做到最佳平衡的语言是最好的选择！

除了“兼顾多方、平衡发展”的特点，Go 语言的另一显著特点是，为并发和并行编程提供极佳的支持。这一点对于更好地利用多核和多处理器计算机非常重要，而且也是 Go 语言最强有力的部分。Go 语言通过 goroutine 这种特有的应用程序线程实现代码片段的并发执行，然后通过 channel 实现各个 goroutine 之间的通信。它弥补了现存编程语言在这方面的不足，但支撑 goroutine 和 channel 的底层原理却很少被人熟知。专题文章《Go 的并发机制》会从线程实现模型、调度器和其他细节部分揭秘 Go 并发机制的背后原理。

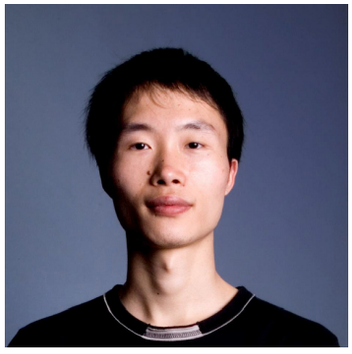
当然，计算机的发展对未来编程语言的要求远非这么简单。专题文章《为什么需要一门新语言》从并发与分布式支持、软件工程支持和编程哲学重塑三方面，详细展示了 Go 语言的应对方案。

“践行”文章分别针对Go测试和Go并发展开讨论。《用正确的方式，打开Go测试》集中介绍了多种测试技巧，对于习惯使用现成的框架来模拟、断言以及做一些其他测试的开发人员来说，“清空脑袋”是首先要做的事。《解决并发之痛》开篇就诊断出“并发之痛”的根源，然后就症状一一给出“处方”。俗话说，“是药三分毒”，不存在完美无副作用的方案，所以作者告诫读者朋友“下方”前，自己斟酌是保险。

Let's Go！ ■

专题：Just Go！

为什么需要一门新语言



作者/许式伟

七牛云存储CEO，曾任盛大创新院资深研究员、金山软件技术总监、WPS Office 2005首席架构师。开源爱好者，发布过包括WINX、TPL等十余个C++开源项目，拥有超过15年的C/C++开发经验。在接触Go语言后就被其大道至简、少即是多的设计哲学所倾倒。七牛云存储是国内第一个吃螃蟹的团队，核心服务完全采用Go语言实现。

现有的编程语言已经很多，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，那么我们为什么还需要Go这样一门新语言呢？

2000年以前的单机时代，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。尽管我们看起来Java已经深获人心，但Java编程的体验并不尽如人意。查看历年来的TIOBE编程语言排行榜就会发现，Java语言的市场份额在逐步下降，显示了这门语言后劲不足。

Go语言的官方文件声称，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C++，想要把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++真的没什么好学的，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。作为互联网时代的C语言，Go必须能够像操作单机一样简单地操作多核计算机与计算机集群。

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。单机时代的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，而是通过库的方式支持协程，但这种功能并不完整，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“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”，其主体思想有两点：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需要考虑的重要课题。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的语句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，以便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，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 {
```

《Go语言编程》以介绍Go语言特性为主，示例多采用七牛云存储团队平常的实践，内容涉及内存管理（堆和栈）、错误处理、OOP、并发编程等关键话题。

这本书面向的读者是所有打算用Go语言的开发者，主要包括目前使用C、C++、Java、C#的开发人员，甚至一些Python、PHP开发人员也可能转为Go程序员。

```
    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。

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。■

专题：Just Go！

Go的线程实现模型

作者/郝林

从业12年有余的软件工匠，国内知名的Go语言技术布道者，Go语言北京用户组和GoHackers社群的发起人和组织者，多套免费在线Go语言教程的作者，深信Go语言在人工智能时代和机器人时代也能大放异彩的科技信徒。

Go不推荐使用共享内存的方式传递数据，而推荐使用channel（或称“通道”）在多个goroutine之间传递数据，同时保证整个过程的并发安全性。不过，作为可选方法，Go依然提供了一些传统的同步方法（比如互斥量、条件变量等）。

在操作系统提供的内核线程之上，Go搭建了一个特有的两级线程模型。我们可以将goroutine看作是Go特有的应用程序线程。但是，goroutine背后的支撑体系远没有这么简单。

说起Go的线程实现模型，有三个必知的核心元素，它们支撑起了这个模型的主框架。

- M：machine的缩写。一个M代表一个内核线程，或者“工作线程”。
- P：processor的缩写。一个P代表执行一个Go代码片段所必需的资源（或称“上下文环境”）。
- G：goroutine的缩写。一个G代表一个Go代码片段。前者是对后者的一种封装。

简单来说，一个G的执行需要P和M的支持。一个M在与一个P关联之后，就形成了一个有效的G运行环境（内核线程+上下文环境）。每个P都会包含一个可运行的G的队列（runq）。该队列中的G会被依次传递给与本地P关联的M，并获得运行时机。在这里，我把运行当前G的那个M称为“当前M”，并把与当前M关联的那个P称为“本地P”。后面我会以此为术语进行描述。

从宏观上看，M、P和G之间的联系如图2-1所示，但是实际关系要比这幅图所展示的关系复杂很多。不过，请先不用理会这里所说的复杂关系，让我们把焦点扩大一些，看看它们与内核调度实体（KSE）之间的关系是怎样的，如图2-2所示。

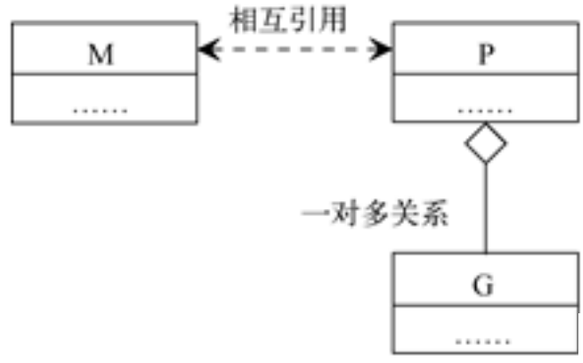


图2-1　GO语言线程实现模型中的三个核心元素

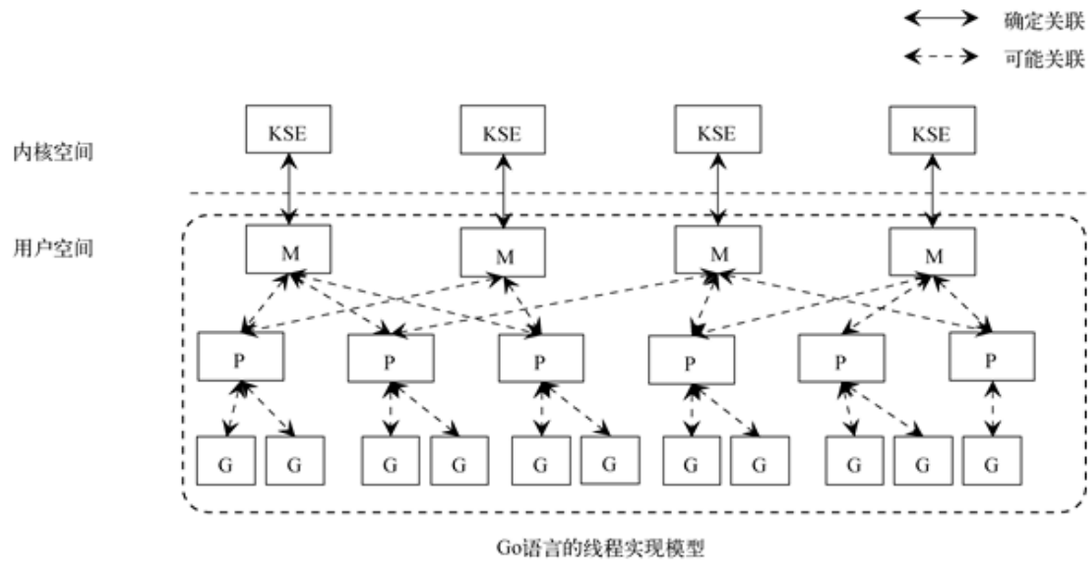


图2-2　M、P、G与KSE的关系

可以看到，M与KSE之间总是一对一的关系，一个M能且仅能代表一个内核线程。Go的运行时系统（runtime system）用M代表一个内核调度实体。M与KSE之间的关联非常稳固，一个M在其生命周期内，会且仅会与一个KSE产生关联。相比之下，M与P、P与G之间的关联都是易变的，它们之间的关系会在实际调度的过程中改变。此外，M与G之间也会建立关联，因为一个G终归会由一个M来负责运行；它们之间的关联会由P来牵线。注意，由于M、P和G之间的关系在实际调度过程中多变，图2-2中的可能关联仅能作为一般性的示意。

至此，你已经知道了这些核心实体之间可能存在的关系。Go的运行时系统会对这些实体的实例进行实时管理和调度。接下来，我会专门对此进行介绍。现在，让我们再次聚焦，看一看在这些实体内部都有哪些值得关注的细节。

M（内核线程）

一个M代表了一个内核线程。在大多数情况下，创建一个新M的原因是没有足够的M来关联P并运行其中可运行的G。不过，在运行时系统执行系统监控或垃圾回收等任务的时候，也会导致新M的创建。M的部分结构如图2-3所示。

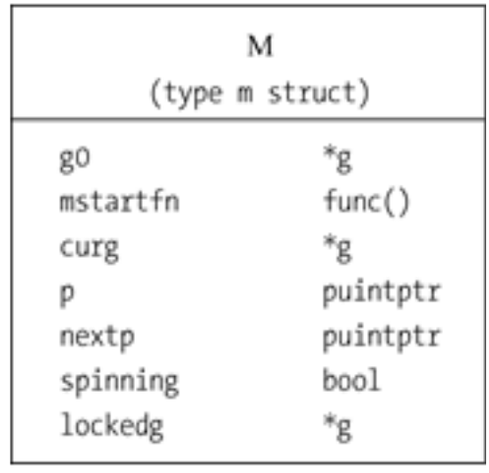


图2-3　M的结构（部分）

M结构中的字段众多，这里只挑选了几个最重要的字段。列表中，每一行都展示了一个字段，左边是字段名，右边是字段类型。其中，字段g0表示一个特殊的goroutine。这个goroutine是Go运行时系统在启动之初创建的，用于执行一些运行时任务。字段mstartfn表示M的起始函数，这个函数其实就是我们在编写go语句时携带的那个函数。字段curg会存放当前M正在运行的那个G的指针，而字段p的值则会指向与当前M相关联的那个P。mstartfn、curg和p最能体现当前M的

即时情况。此外，字段nextp用于暂存与当前M有潜在关联的P。让调度器将某个P赋给某个M的操作由nextp字段控制，称为对M和P的预联。运行时系统有时候会把刚刚重新启用的M和已与它预联的那个P关联在一起，这也是nextp字段的主要作用。字段spinning是bool类型的，它用于表示这个M是否正在寻找可运行的G。在寻找过程中，M会处于自旋状态。这也是该字段名的由来。Go运行时系统可以把一个M和一个G锁定在一起。一旦锁定，这个M就只能运行这个G，这个G也只能由该M运行。标准库代码包runtime中的函数LockOSThread和UnlockOSThread，也为我们提供了锁定和解锁的具体方法。M的字段lockedg表示的就是与当前M锁定的那个G（如果有的话）。

M在创建之初，会被加入全局的M列表（runtime.allm）中。这时，它的起始函数和预联的P也会被设置。最后，运行时系统会为这个M专门创建一个新的内核线程并与之相关联。如此一来，这个M就为执行G做好了准备。其中，起始函数仅当运行时系统要用此M执行系统监控或垃圾回收等任务的时候才会被设置。而这里的全局M列表其实并没有什么特殊的意义。运行时系统在需要的时候，会通过它获取到所有M的信息。同时，它也可以防止M被当作垃圾回收掉。

在新M被创建之后，Go运行时系统会先对它进行一番初始化，其中包括对自身所持的栈空间以及信号处理方面的初始化。在这些初始化工作都完成之后，该M的起始函数会被执行（如果存在的话）。注意，如果这个起始函数代表的是系统监控任务的话，那么该M会一直执行它，而不会继续后面的流程。否则，在起始函数执行完毕之后，当前M将会与那个预联的P完成关联，并准备执行其他任务。M会依次在多处寻

找可运行的G并运行它。这一过程也是调度的一部分。有了M，Go程序的并发运行基础才得以形成。

运行时系统管辖的M（或者说runtime.allm中的M）有时候也会被停止，比如在运行时系统执行垃圾回收任务的过程中。运行时系统在停止M的时候，会把它放入调度器的空闲M列表（runtime.sched.midle）。这很重要，因为在需要一个未被使用的M时，运行时系统会先尝试从该列表中获取。M是否空闲，仅以它是否存在于调度器的空闲M列表中为依据。

单个Go程序所使用的M的最大数量是可以设置的。Go程序运行的时候会先启动一个引导程序，这个引导程序会为其运行建立必要的环境。在初始化调度器的时候，它会对M的最大数量进行初始设置，这个初始值是10 000。也就是说，一个Go程序最多可以使用10 000个M。这就意味着，最多可以有10 000个内核线程服务于当前的Go程序。请注意，这里说的是最理想的情况；由于操作系统内核对进程的虚拟内存的布局控制以及大小限制，如此量级的线程可能很难共存。从这个角度看，Go本身对于线程数量的限制几乎可以忽略。

除了上述初始设置之外，我们也可以在Go程序中对该限制进行设置。为了达到此目的，你需要调用标准库代码包runtime/debug中的SetMaxThreads函数，并提供新的M最大数量。runtime/debug.SetMaxThreads函数在执行完成后，会把旧的M最大数量作为结果值返回。非常重要的一点是，如果你在调用runtime/debug.SetMaxThreads函数时给定的新值比当时M的实际数量还要小，运行时系统就会立即

引发一个运行时恐慌。所以，你要非常谨慎地使用这个函数。请记住，如果真的需要设置M的最大数量，那么越早调用runtime/debug.SetMaxThreads函数越好。对于它的设定值，你也要仔细斟酌。

P（执行一个Go代码片段所必需的资源）

P是G能够在M中运行的关键。Go的运行时系统会适时地让P与不同的M建立或断开关联，以使P中的那些可运行的G能够及时获得运行时机，这与操作系统内核在CPU之上实时地切换不同的进程或线程的情形类似。

改变单个Go程序间接拥有的P的最大数量有两种方法。第一种方法，调用函数runtime.GOMAXPROCS并把想要设定的数量作为参数传入。第二种方法，在Go程序运行前设置环境变量GOMAXPROCS的值。P的最大数量实际上是对程序中并发运行的G的规模的一种限制。P的数量即为可运行G的队列的数量。一个G在被启用后，会先被追加到某个P的可运行G队列中，以等待运行时机。一个P只有与一个M关联在一起时，才会使其可运行G队列中的G有机会运行。不过，设置P的最大数量只能限制住P的数量，而对G和M的数量没有任何约束。当M因系统调用而阻塞（更确切地说，是它运行的G进入了系统调用）的时候，运行时系统会把该M和与之关联的P分离开来。这时，如果这个P的可运行G队列中还有未被运行的G，那么运行时系统就会找到一个空闲M，或创建一个新的M，并与该P关联以满足这些G的运行需要。因此，M的数量在很多时候也都会比P多。而G的数量，一般取决于Go程序本身。

在Go程序启动之初，引导程序会在初始化调度器时，对P的最大数量进行设置。这里的默认值会与当前CPU的总核心数相同。一旦发现环境变量GOMAXPROCS的值大于0，引导程序就会认为我们想要对P的最大数量进行设置。它会先检查一下此值的有效性：

如果不大于预设的硬性上限值（256），就认为是有效的，否则就会被这个硬性上限值取代。也就是说，最终的P的最大数量值绝不会比硬性上限值大。硬性上限值是256的原因是，Go目前还不能保证在256多个P同时存在的情形下仍然保持高效。不过，这个硬性上限值并不是永久的，它可能会在未来改变。

注意，虽然Go并未对何时调用runtime.GOMAXPROCS函数作限制，但是该函数调用的执行会暂时让所有的P都脱离运行状态，并试图阻止任何用户级别的G的运行。只有在新的P最大数量设定完成之后，运行时系统才开始陆续恢复它们。这对于程序的性能是非常大的损耗。所以，你最好只在Go程序的main函数的最前面调用runtime.GOMAXPROCS函数。当然，不在程序中改变P的最大数量最好不过，实际上在大多数情况下也无需改变。

确定P的最大数量之后，运行时系统会根据这个数值重整全局的P列表（runtime. allp）。与全局M列表类似，该列表中包含了当前运行时系统创建的所有P。运行时系统会把这些P中的可运行G全部取出，并放入调度器的可运行G队列中。这是调整全局P列表的一个重要前提。被转移的那些G，会在以后经由调度再次放入某个P的可运行G队列。

与空闲M列表类似，运行时系统中也存在一个调度器的空闲P列表（runtime.sched.pidle）。当一个P不再与任何M关联的时候，运行时系统就会把它放入该列表；而当运行时系统需要一个空闲的P关联某个M的话，会从此列表中取出一个。注意，P进入空闲P列表的一个前提条件是它的可运行G列表必须为空。例如，在重整全局P列表的时候，P在被清空可运行G队列之后，才会被放入空闲P列表。

与M不同，P本身是有状态的，可能具有的状态如下。

- Pidle 此状态表明当前P未与任何M存在关联。
- Prunning 此状态表明当前P正在与某个M关联。
- Psyscall 此状态表明当前P中的运行的那个G正在进行系统调用。
- Pgcstop 此状态表明运行时系统需要停止调度。例如，运行时系统在开始垃圾回收的某些步骤前，就会试图把全局P列表中的所有P都置于此状态。
- Pdead 此状态表明当前P已经不会再被使用。如果在Go程序运行的过程中，通过调用runtime.GOMAXPROCS函数减少了P的最大数量，那么多余的P就会被运行时系统置于此状态。

P在创建之初的状态是Pgcstop，虽然这并不意味着运行时系统要在这时进行垃圾回收。不过，P处于这一初始状态的时间会非常短暂。在紧接着的初始化之后，运行时系统会将其状态设置为Pidle，并放入调度器的空闲P列表。图2-4描绘了P在各个状态之间进行流转的具体情况。

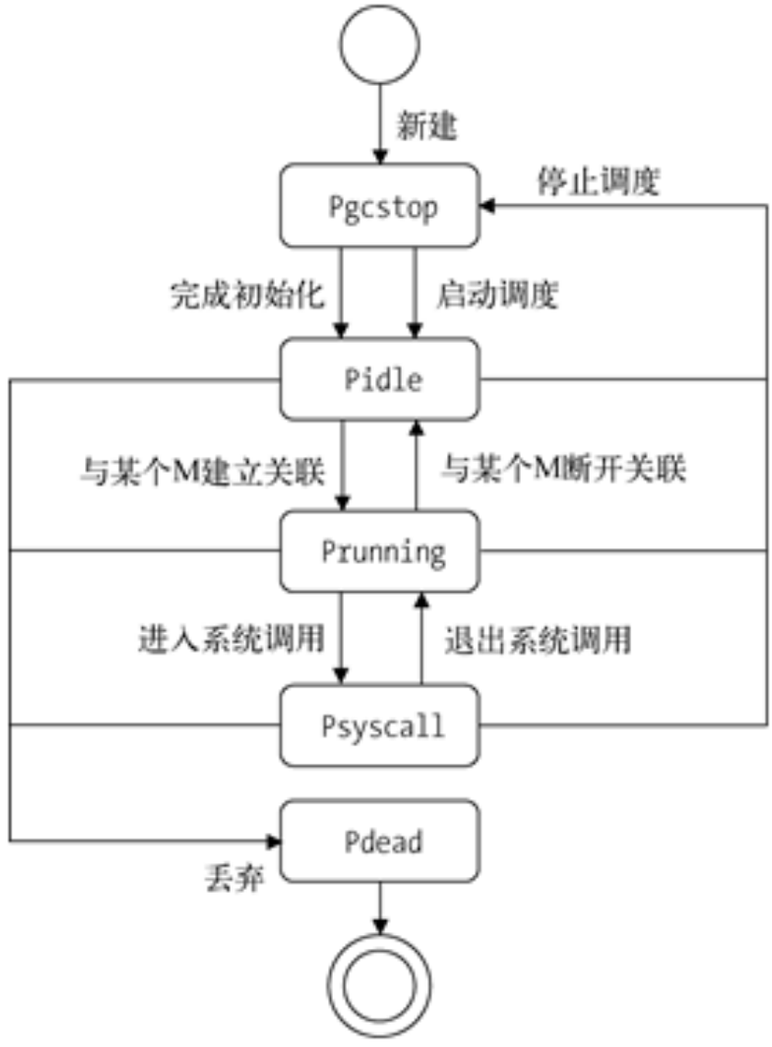


图2-4　P的状态转换

可以看到，非Pdead状态的P都会在运行时系统欲停止调度时被置于Pgcstop状态。不过，等到需要重启调度的时候（如垃圾回收结束后），它们并不会被恢复至原有状态，而会被统一地转换为Pidle状态。也就是说，它们会被放到同一起跑线上，并公平地接受再次调度。另一方面，

非Pgcstop状态的P都可能因全局P列表的缩小而被认为是多余的，并被置于Pdead状态。不过，我们并不用担心其中的G会失去归宿。因为，在P被转换为Pdead状态之前，其可运行G队列中的G都会被转移到调度器的可运行G队列，而它的自由G列表中的G也都会被转移到调度器的自由G列表中。

正如前面所述，每个P中除了一个可运行G队列外，还都包含一个自由G列表。这个列表中包含了一些已经运行完成的G。随着运行完成的G的增多，该列表可能会很长。如果它增长到一定程度，运行时系统就会把其中的部分G转移到调度器的自由G列表中。另一方面，当使用go语句启用一个G的时候，运行时系统会先试图从相应P的自由G列表中获取一个现成的G，来封装这个go语句携带的函数（也称go函数），仅当获取不到这样一个G的时候才有可能创建一个新的G。考虑到相应P的自由G列表为空而获取不到自由G的情况，运行时系统会在发现其中的自由G太少时，预先尝试从调度器的自由G列表中转移过来一些G。如此一来，只有在调度器的自由G列表也弹尽粮绝的时候，才会有新的G被创建。这在很大程度上提高了G的复用率。

在P的结构中，可运行G队列和自由G列表是最重要的两个成员。至少对于Go语言的使用者来说是这样。它们间接地体现了运行时系统对G的调度情况。下面就对Go并发模型中的G进行介绍。

G（Go代码片段）

一个G就代表一个goroutine（或称Go例程），也与go函数相对应。作为编程人员，我们只是使用go语句向Go的运行时系统提交了一个并发任务，而Go的运行时系统则会按照我们的要求并发地执行它。

Go的编译器会把go语句变成对内部函数newproc的调用，并把go函数及其参数都作为参数传递给这个函数。这也是你应该了解的第一件与go语句有关的事。其实它并不神秘，只是一种递送并发任务的方法而已。

运行时系统在接到这样一个调用之后，会先检查go函数及其参数的合法性，然后试图从本地P的自由G列表和调度器的自由G列表获取可用的G，如果没有获取到，就新建一个G。与M和P相同，运行时系统也持有一个G的全局列表（runtime.allgs）。新建的G会在第一时间被加入该列表。类似地，这个全局列表的主要作用是：集中存放当前运行时系统中的所有G的指针。无论用于封装当前这个go函数的G是否是新创建的，运行时系统都会对它进行一次初始化，包括关联go函数以及设置该G的状态和ID等步骤。在初始化完成后，这个G会立即被存储到本地P的runnext字段中；该字段用于存放新鲜出炉的G，以求更早地运行它。如果这时runnext字段已存有一个G，那么这个已有的G就会被“踢到”该P的可运行G队列的末尾。如果该队列已满，那么这个G就只能追加到调度器的可运行G队列中了。

在特定情况下，一旦新启用的G被存于某地，调度就会立即进行以使该G尽早被运行。不过，即使这里不立即调度，我们也无需担心，因为运行时系统总是在为及时运行每个G忙碌着。

每一个G都会由运行时系统根据其实际状况设置不同的状态，其主要状态如下。

- Gidle 表示当前G刚被新分配，但还未初始化。
- Grunnable 表示当前G正在可运行队列中等待运行。
- Grunning 表示当前G正在运行。
- Gsyscall 表示当前G正在执行某个系统调用。
- Gwaiting 表示当前G正在阻塞。
- Gdead 表示当前G正在闲置。
- Gcopystack 表示当前G的栈正被移动，移动的原因可能是栈的扩展或收缩。

除了上述状态，还有一个称为Gscan的状态。不过这个状态并不能独立存在，而是组合状态的一部分。比如，Gscan与Grunnable组合成Gscanrunnable状态，代表当前G正等待运行，同时它的栈正被扫描，扫描的原因一般是GC（垃圾回收）任务的执行。又比如，Gscan与Grunning组合成Gscanrunning状态，表示正处于Grunning状态的当前G的栈要被GC扫描时的一个短暂时刻。简单起见，我不会在下面对这些组合状态进行说明。你只要知道这些组合状态会在GC扫描发生时出现就可以了。

之前讲过，在运行时系统想用一个G封装go函数的时候，会先对这个G进行初始化。一旦该G准备就绪，其状态就会被设置成Grunnable。

《Go并发编程实战2》首先介绍了Go语言的优秀特性、安装设置方法、工程结构、标准命令和工具、语法基础、数据类型以及流程控制方法，接着阐述了与多进程编程和多线程编程有关的知识，然后重点介绍了goroutine、channel以及Go提供的传统同步方法，最后通过一个完整实例——网络爬虫框架进一步阐述Go语言的哲学和理念，同时分享作者在多年编程生涯中的一些见解和感悟。与上一版相比，本书不仅基于Go 1.8对上一版进行了全面更新，而且更深入地描绘了Go运行时系统的内部机理，并且大幅改进了示例代码。

也就是说，一个G真正开始被使用是在其状态设置为Grunnable之后。图2-5展示了G在其生命周期内的状态流转情况。

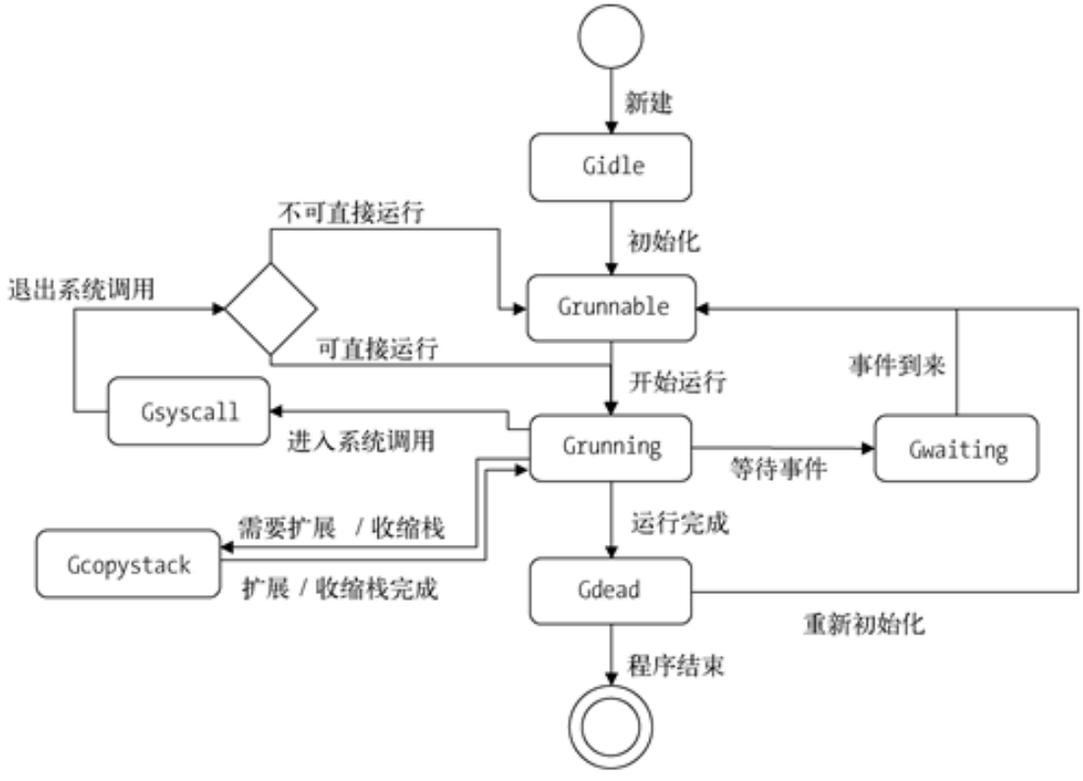


图2-5　G 的状态转换

一个G在运行的过程中，是否会等待某个事件以及会等待什么样的事件，完全由其封装的go函数决定。例如，如果这个函数中包含对通道值的操作，那么在执行到对应代码的时候，这个G就有可能进入Gwaiting状态。这可能是在等待从通道类型值中接收值，也可能是在等待向通道类型值发送值。又例如，涉及网络I/O的时候也会导致相应的G进入Gwaiting状态。此外，操纵定时器（time.Timer）和调用time.Sleep函

数同样会造成相应G的等待。在事件到来之后，G会被“唤醒”并被转换至Grunnable状态。待时机到来时，它会被再次运行。

G在退出系统调用时的状态转换要比上述情况复杂一些。运行时系统会先尝试直接运行这个G，仅当无法直接运行的时候，才会把它转换为Grunnable状态并放入调度器的自由G列表中。显然，对这样一个G来说，在其退出系统调用之时就立即被恢复运行再好不过了。运行时系统当然会为此做出一些努力，不过即使努力失败了，该G也还是会在实时的调度过程中被发现并运行。

最后，值得一提的是，进入死亡状态（Gdead）的G是可以重新初始化并使用的。相比之下，P在进入死亡状态（Pdead）之后，就只能面临销毁的结局。由此也可以说明Gdead状态与Pdead状态所表达的含义截然不同。处于Gdead状态的G会被放入本地P或调度器的自由G列表，这是它们被重用的前提条件。

至此，你已经基本了解了一个G在Go运行时系统中的流转方式和状态转换时机。

我一直把实现和操纵Go的线程实现模型的内部程序笼统地称为“运行时系统”。实际上，它可以更明确地称为“调度器”。一个Go程序中只会存在一个调度器实例。它拥有自己的结构，同时依次提供了很多重要的运行时功能。其中一些功能其实已经讲到了。只不过，我是围绕着各种模型元素（M、P、G以及各种容器）讲解的。■

专题：Just Go！

助力谷歌App Engine云平台

作者/樊虹剑（fango）

2006年，亚马逊发现自己在因特网泡沫时大量投资的服务器，竟然只有一成的利用率，就推出产品AWS，让它的客户有办法通过标准的HTTP协议，租用共享这些设备提供的服务。谷歌也不甘落后，在2008年发布App Engine，并引发了业界的狂热。

云的由来，传统的说法是因特网模型在纸上画不出来，就随便圈几笔代表抽象的远端共享的意思。据说这也继承了20世纪初老式电话系统POTS的传统。的确如此，电话发明以后，人类首次实现了不受时间和空间约束的信息实时传播。它彻底改变了人们的通信方法和生活方式，也造就了美国电报电话以及各国电信企业的垄断局面。从这里我们也可以引申出云计算的特征。

1. 信息高度集中在运营商的数据中心。

2. 用户无法也无需关心自己信息的保存。

3. 使用唯一的通信标准保障互联互通。

4. 终端廉价，随手可得，容易使用。

5. 按信息以及软硬件的使用收费。

对普通使用者而言，云计算就是浏览网页。只是除了被动地获取影音文字，也可以主动地提供信息和搜索信息。尽管背后的软件一如既往地处理这些信息，但使用方式的改变，让普通用户跳出了传统的购买、安装、培训、使用软件的模式。也就是说软件自身改变不大，只是对用户变得透明了。让他们使用习惯的浏览方式，去避免和目标无关的麻烦。

这听起来非常棒。但对软件开发者的挑战在于，它是虚的东西，关乎的是用户体验，而不像传统的窗口应用那样千人一面。这又回到了我们一贯的主题：程序之所以也叫“设计”，是因为它是大脑构思的产物，是设计师自身修为的体现。程序设计和用户体验的设计是一脉相通的东西。希望读者不要满足于掌握某种语言的相关技术，而应该不懈地去追求程序设计本身的这种美。

现有的云计算可分为三种，从低到高分别称为：IaaS、Paas 和 SaaS。也就是基础服务（Infrastructure as a Service）、平台服务（Platform as a Service）和软件服务（Software as a Service）。基础服务是亚马逊 AWS 的模式，只出租服务器；软件服务是 salesforce.com 和 Google Apps 这样的互联网软件寻租模式；平台服务是我们要详谈的谷歌 App Engine 的模式，也就是提供一个开发和运行环境，使程序员不必像基础服务那样事必躬亲，也不会像软件服务那样没有机会动手。

GAE

谷歌的 App Engine 简称 GAE，它可以直接使用谷歌在全球的数据中心，用 Go 语言编写因特网应用程序。因为这些 App 使用的软硬件系统和谷

歌自己网站的系统是一样的，所以无论从安全性、性能、可靠性还是可扩展性方面，都比自行架设的网站服务器更胜一筹。无论是个人主页这种极小规格的应用，还是几十万人的跨国公司的内部作业系统，GAE提供的云计算平台服务都可以胜认。特别是对从一粒种子能够迅速成长为参天大树的因特网起步公司，GAE云平台极强的扩展能力，更是不容忽视。

GAE的Go处理器是Go语言在谷歌内部运行一年后，对外发布的第一个大规模的应用项目。它也是目前Go语言最吸引程序员的项目之一。App的创建、使用和维护都很简单：GAE的开发者把Go写的代码传上去，申请一个URL指向它，就可以用浏览器运行那些代码提供的服务了。而同样使用浏览器指向App的管理网页，就可以监督此App的数据流量、存储空间和CPU使用情况。这些都是App的收费项目。对于不大的App，GAE是免费的；而对于大规模应用，GAE便可为谷歌带来大笔收入。尽管收费了，但因为所有软硬件和服务都是全球共享的。除非某家公司能达到谷歌的规模，或者有这个梦想以及财力，不然，比起独立运作自己的环球因特网应用系统平台，共享还是会节省很多钱。这种分工合作的规模经济，是有别于小农经济的现代商业的核心，也是云计算一待成熟就一飞冲天的原因。

由于在GAE上，用Go编译的程序运行在谷歌的服务器上，所以此时Go的使用方式，与在自己管理的计算机上使用的Go标准包，有一些区别。具体情况如下。

1. 需要使用App Engine提供的Go SDK，而不是标准的Go下载包。

2. App使用的是类似Go的http包的界面；而Go的App类似于标准的Go网站服务器。

3. App运行时自动编译，不需直接使用Go编译器。

4. App运行在沙盒里，不可以使用某些需要特权的Go包。例如，文件和套接字（socket）都不可以直接使用。

5. App可以使用的资源受配额quota的限制。

总之，在GAE服务平台上使用Go，很具吸引力也很容易。接下来，先简要分析Go SDK自带的几个演示程序，让读者尽快熟悉、尽快入门。

Hello 世界！

开始写云代码的第一步，当然是下载一个软件开发包。从https://developers.google.com/appengine/downloads，可以得到Linux、Mac和Windows上的SDK压缩包。

展开压缩包，进入目录，遵照传统方式执行第一个打招呼程序：

```
./dev_appserver.py demos/helloworld
```

如果一切ok，它会说：到http://localhost:8080 看结果。那我们就要用浏览器了。接下来逐行审视一下这个打招呼程序：

```
package helloworld

import (
```

```
    "fmt"
    "net/http"
)

func init() {
    http.HandleFunc("/", handle)
}

func handle(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "<html><body>Hello, World!      !</body></html>")
}
```

程序的入口是 init，这是因为 Go 的 main 函数是 GAE 提供的。handle 中的浏览器访问请求由 r 指向，回复写入 w 缓冲区，在 handle 返回时，交由 GAE 传送给浏览器。

为了让 GAE 能正确找到我们的程序，在 helloworld 目录下要准备一个叫 app.yaml 的配置文件，其主要内容为：

```
runtime: go
```

它指定 GAE 运行的是 Go 处理器。

```
handlers:
- url: /.*
  script: _go_app
```

告诉 GAE 把 / 下所有路径都交由 _go_app 处理。小心不要拼错，否则会出现一堆 Python 错误。

这个 _go_app 位于 google/appengine/ext/go/ 目录下的 _init_.py 里，它是架通 _dev_appserver.py 和我们的 Go 程序的桥梁。当浏览器指向我们的

URL时，_dev_appserver会编译我们的Go程序，并存放在/tmp下新创建的一个目录里，文件名就叫_go_app。这是个货真价实的可执行文件，所有用到的库和运行环境全部打包，一应俱全，不存在外部依赖。

/tmp下还有两个文件用于通信，读者可以自己查看源代码。

画胡子

这里我们回顾一下Andrew和Rob在Google IO大会上演示的moustach-io，了解标准Go网站的程序如何在GAE上运行。当然，我们还是用dev_appserver来冒充真正的云计算。

```
import (
    "bytes"
    "encoding/json"
    "errors"
    "fmt"
    "image"
    "image/jpeg"
    _ "image/png"
    "io"
    "net/http"
    "strconv"
    "text/template"

    "code.google.com/p/goauth2/oauth"
)
```

示例中的图像压缩使用JPEG的函数，所以会导入image/jpeg包。image.Decode图像解压也可以使用PNG格式，但Go不允许导入没用到其函数的包。由于此程序没有png的函数，所以用_（空白标识符）来导入image/png包，用init函数注册decode需要的PNG格式。

goauth2和freetype（在draw.go中）是独立于GAE的第三方库，读者可以参考README，用hg安装以便能编译到_go_app的可执行代码中。

```
func init() {
    http.HandleFunc("/", errorHandler(upload))
    http.HandleFunc("/edit", errorHandler(edit))
    http.HandleFunc("/img", errorHandler(img))
    http.HandleFunc("/share", errorHandler(share))
    http.HandleFunc("/post", errorHandler(post))
    editTemplate = template.New(nil)
    editTemplate.SetDelims("{{{", "}}}")
    if err := editTemplate.ParseFiles("edit.html"); err != nil {
        panic("can't parse edit.html: " + err.Error())
    }
}
```

入口注册URL路径的处理函数，editTemplate的分隔符设定为3个{}。errorHandler很有趣，我们一起看一下。

```
func errorHandler(fn http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        defer func() {
            if err, ok := recover().(error); ok {

    w.WriteHeader(http.StatusInternalServer Error)
                errorTemplate.Execute(w, err)
            }
        }()
        fn(w, r)
    }
}

// check aborts the current execution if err is non-nil.
func check(err error) {
    if err != nil {
```

```
        panic(err)
    }
}
```

这里巧妙地使用了Go语言提供的panic，defer和recover的异常处理方式。我们可以这样看：当check的err非空时，panic导致check及其调用栈的所有函数（例如upload和errorHandler）立即退出。但errorHandler定义了defer，会在退出前执行。而其使用的recover，会取得panic的变量，即err。这里使用类型断言确定它是error类型，并在此处恢复panic引起的退出，然后用errorTemplate在浏览器中显示出错信息。如果没有panic，或panic传入的不是error类型的值，defer则不执行任何操作。

有没有注意到？errorHandler接受一个函数，包装defer后，返回一个无名函数。Go的函数可以像普通变量一样，等一下我们会看到。

```
type Image struct {
    Data []byte
}
```

定义Image结构体，其Data项是字节切片，用来方便安全地完成类似C的指针操作。

```
func upload(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        // No upload; show the upload form.
        uploadTemplate.Execute(w, nil)
        return
    }
    f,_,err := r.FormFile("image")
    check(err)
    defer f.Close()
```

defer是在函数退出时执行的。它紧跟在打开文件之后，以防忘记关闭。并且每个err都用check进行异常处理。

```
var buf bytes.Buffer
io.Copy(&buf, f)
i,_,err := image.Decode(&buf)
check(err)
```

bytes.Buffer自动管理字节内存，io.Copy可以高效地从文件f复制到buf。image.Decode能分析图像格式，并自动用jpeg或png包的Decode函数解压。

```
   const max = 1200
if b := i.Bounds(); b.Dx() > max || b.Dy() > max {
```

if语句可以有初始赋值子句。此处的变量b只在if后的块里有效，即其作用域由{}限定。

```
   if b.Dx() > 2*max || b.Dy() > 2*max {
    w, h := max, max
    if b.Dx() > b.Dy() {
        h = b.Dy() * h / b.Dx()
   } else {
        w = b.Dx() * w / b.Dy()
    }
    i = resize.Resample(i,i.Bounds(), w, h)
    b = i.Bounds()
}
w, h := max/2, max/2
if b.Dx() > b.Dy() {
    h = b.Dy() * h / b.Dx()
} else {
    w = b.Dx() * w / b.Dy()
```

```
}
i = resize.Resize(i, i.Bounds(), w, h)
}
```

由于局部作用域的关系，变量名可以很短，同时我们不用担心会和作用域外的同名变量起冲突。

```
// Encode as a new JPEG image
   buf.Reset()
   err = jpeg.Encode(&buf, i, nil)
   check(err)
// Create an App Engine Context for the client's request.
   c := appengine.NewContext(r)
// Save the image under a unique key,a hash of the image.
   key := datastore.NewKey(c, "Image", keyOf(buf.Bytes()), 0, nil)
   _,err = datastore.Put(c,key,&Image{buf.Bytes()})
   check(err)
//Redirect to/edit using the key.
   http.Redirect(w,r,"/edit?id="+key.StringID(), http.StatusFound)
}
```

用r的FromFile通过浏览器下载上传的图像，解压检查，重新采样改变大小，再压缩回JPEG格式，放入App Engine的datastore，再让浏览器重定向到edit页面，添加大胡子。

```
func edit(w http.ResponseWriter,r *http.Request) {
    editTemplate.Execute(w,r.FormValue("id"))
}
```

在init()中指定处理URL /edit的请求，简单地执行对应的模版。

```
func keyOf(data []byte) string {
    sha := sha1.New()
    sha.Write(data)
    return fmt.Sprintf("%x",string(sha.Sum())[0:8])
}
```

由 sha1 得到对应数据的唯一指纹，注意，API 是这么地简单，一写一读即告完成。

```
func img(w http.ResponseWriter, r *http.Request) {
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Image", r.FormValue("id"), 0,nil)
    im := new(Image)
    err := datastore.Get(c, key, im)
    check(err)
    m,_,err := image.Decode(bytes.NewBuffer(im.Data))
    check(err)
    get := func(n string) int { // helper closure
        i, _ := strconv.Atoi(r.FormValue(n))
        return i
    }
    x,y,s,d := get("x"),get("y"),get("s"),get("d")
```

这里用到了闭包。get 是只限此处使用的函数，其内部的 r 从外部的函数传入（闭包），n 和 i 是正常的参数和返回值。

```
if x > 0 { // only draw if coordinates provided
   m = moustache(m, x, y, s, d)
}
```

moustache 在 draw.go 中定义，也使用 package moustachio，编译器会把它们一起编译。

```
w.Header().Set("Content-type", "image/jpeg")
jpeg.Encode(w, m, nil)
}
```

然后把 jpeg 图像传回浏览器。

```
func share(w http.ResponseWriter, r *http.Request) {
```

```
    url := config(r.Host).AuthCodeURL(r.URL.RawQuery)
    http.Redirect(w, r, url, http.StatusFound)
}
```

留言录

说起真正用到谷歌云平台API的程序，就要谈到留言录（guestbook）。在Go语言里，这些API是包装在appengine包里的公共函数、常量、变量。我们从头看起：

```
package guestbook

import (
    "io"
    "net/http"
    "text/template"
    "time"

    "appengine"
    "appengine/datastore"
    "appengine/user"
)
```

GAE的appengine包、datastore包和user包是比较常用的3个包，分别负责提供Go运行的环境（context），数据库存取，和访问用户的登录。

```
type Greeting struct {
    Author  string
    Content string
    Date    time.Time
}
```

内存中的数据结构会用来读写数据库。

```
func serve404(w http.ResponseWriter) {
    w.WriteHeader(http.StatusNotFound)
    w.Header().Set("Content-Type", "text/plain; charset=utf-8")
    io.WriteString(w, "Not Found")
}
func serveError(c appengine.Context, w http.Response- Writer, err error) {
    w.WriteHeader(http.StatusInternalServerError)
    w.Header().Set("Content-Type", "text/plain; charset=utf-8")
    io.WriteString(w, "Internal Server Error")
    c.Errorf("%v", err)
}
```

serve404和serveError这两个函数是出错时的信息处理函数。

```
var mainPage = template.Must(template.New ("guestbook").Parse(
    `<html><body>
{{range .}}
{{with .Author}}<b>{{.|html}}</b>{{else}}An anonymous person {{end}}
on <em>{{.Date.Format "3:04pm, Mon 2 Jan"}}</em>
wrote <blockquote>{{.Content|html}}</blockquote>
{{end}}
<form action="/sign" method="post">
<div>
<textarea name="content" rows="3" cols="60"> </textarea></div>
<div><input type="submit" value="Sign Guestbook"> </div>
</form></body></html>
`))
```

主页除了逐一显示已有的每一条留言，也提供一个HTML表让用户提交新留言。

```
func handleMainPage(w http.ResponseWriter, r *http.Request) {
    if r.Method != "GET" || r.URL.Path != "/" {
        serve404(w)
        return
    }
    c := appengine.NewContext(r)
```

该函数用于处理主页的请求。我们会为每一个新的http请求分配新的上下文环境c。之后用这个c来把GAE的服务对应到请求上。

```
q := datastore.NewQuery("Greeting").Order("-Date"). Limit(10)
```

可以理解为类似普通数据库的SQL查询，这里是谷歌分布式数据库所用的查询方式：查找Greeting对象，降序排列，限制最多返回10个结果。

```
var gg []*Greeting
_, err := q.GetAll(c, &gg)
if err != nil {
    serveError(c, w, err)
    return
}
```

执行查询，结果放入gg切片。注意，我们无需自己分配gg。

```
w.Header().Set("Content-Type", "text/html; charset=utf-8")
if err := mainPage.Execute(w, gg); err != nil {
    c.Errorf("%v", err)
}
```

显示主页，也就是在浏览器中显示查询结果，以及新留言的提交表。

```
func handleSign(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        serve404(w)
        return
    }
    c := appengine.NewContext(r)
    if err := r.ParseForm(); err != nil {
        serveError(c, w, err)
        return
    }
```

《Go语言·云动力》是Go语言程序设计的入门书。该书介绍了Go语言的基础知识，包括静态类型、流程控制、函数、动态类型、面向对象、并发编程等内容，以及同其他C类语言相比，所具备的全新特性。同时还介绍了Go语言在云计算中的应用。

处理留言表的提交请求。同样每个请求都使用一个新的GAE环境c。

```
g := &Greeting{
    Content: r.FormValue("content"),
    Date:    time.Now(),
}
```

自动分配一个新的Greating结构，填入提交的内容。

```
if u := user.Current(c); u != nil {
    g.Author = u.String()
}
```

如果用户还未登录，Current返回nil，要求用户登录。否则主页返回一个用户结构。我们把它填入Greeting的作者一栏。

```
if _, err := datastore.Put(c, datastore.NewIncompleteKey(c, "Greeting", nil), g);
err != nil {
    serveError(c, w, err)
    return
}
```

无论用户有没有登录，我们都把这个新的Greeting结构写入数据库。NewIncompleteKey让数据库自动分配一个新的key给这条新留言。

```
    http.Redirect(w, r, "/", http.StatusFound)
}
```

返回浏览器，让它重新访问Redirect主页。

```
func init() {
    http.HandleFunc("/", handleMainPage)
    http.HandleFunc("/sign", handleSign)
}
```

GAE 里的 Go 语言使用 `init` 来定义 http 的 URL 路径处理函数。

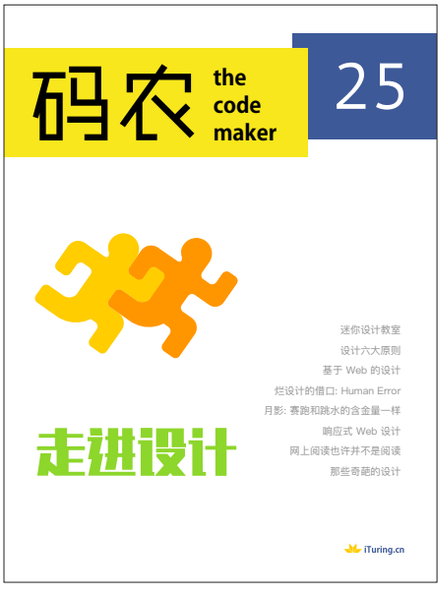
这里我们简要介绍了 Go 语言在云计算方面的几个小知识，如果大家能够顺势应用在云计算上，会发现 Go 在云领域有着极大的潜力，也必会成为事业腾飞的助力。■

读《码农》吐吐槽 还能赚银子！

《码农》电子刊如今慢慢悠悠已经出版了27期，从一开始的好评如潮，到现在的“怎么这么抽象啊”“赶脚内容没有以前好了？”“一下就翻完了，没什么内容啊”……小编表示压力很大，现在的码农读者太难伺候了，明明是本免费杂志，￥%#%&*%@#￥*@￥#@！

但是，一看到好评，编者还是会热泪盈眶，热血沸腾，各种正能量啊“这么好质量的杂志还免费，天上掉馅饼呀”“希望一直出！每一版都读了，很值得推荐！！”……

所以《码农》还是会一直做下去，但是，是好是坏，您得给个话儿啊！



活动规则：在吐槽贴中留言，选出任何一期中你最喜欢的文章和最不喜欢的文章，即可获得图灵社区银子2两！凡吐槽吐得掷地有声者，加赠银子3两（共5两）！（怎样使用银子兑换图书）

活动时间：本活动长期有效。

白宸：

Redis带你尽享丝滑！

访谈嘉宾：

本名郑明杭，现阿里云NoSQL数据库技术专家。先后从事Tair分布式系统、Memcached云服务及阿里云Redis数据库云服务开发，关注分布式系统及NoSQL存储技术前沿。

作为嘉宾，曾在飞马网&中生代技术嘉年华上发表题为《ApsaraDB for Redis云数据库技术栈详解》的演讲内容，获得观众一致叫好！

邀请白宸做客，我们主要聊了些：

- 从事数据库工作的原因
- 阿里云Redis数据库跟其他友商相比的优势
- Redis最得意和最痛的点
- 分布式系统存储技术的前沿——NewSQL
- 数据库实现的方法
- ……

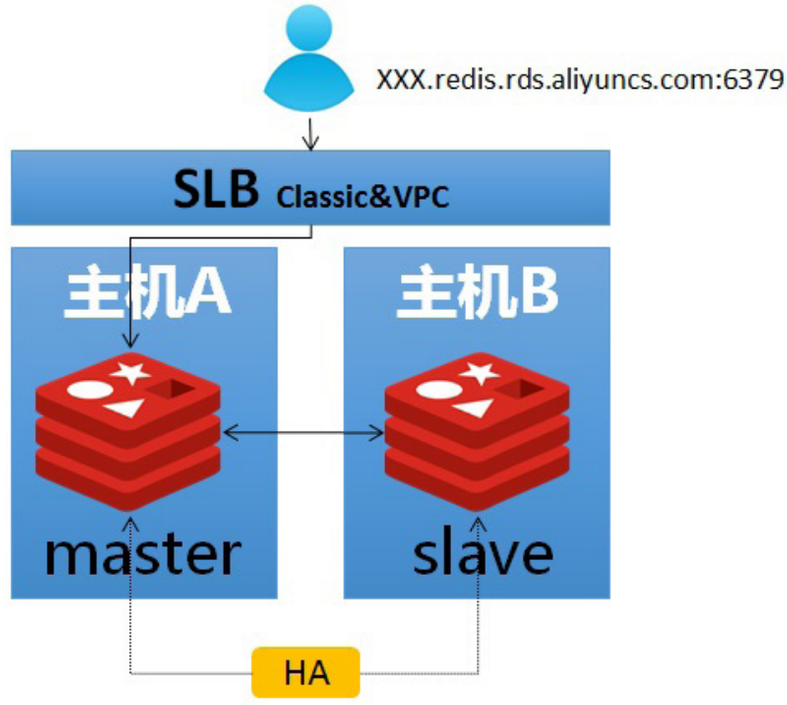
图灵访谈：为什么选择数据库方面的工作？

数据库是产品应用的基础。直播、共享单车等每个火爆产品的背后都是大量关系型数据库、NoSQL类型数据库在支撑。另外，中小企业为了专注自身业务的发展都会选择云计算，将基础设置的建设交给云服务商。云服务商为了提供高性能、高可用的数据库产品需要投入足够多的研发精力。在数据库研发过程中，数据库研发者既可以深入内核、网络、存储设备进行深度的优化改进，又能够在数据库层面扩展出分布式、异地容灾等不同的产品形态，同时结合不同类型的业务进行不同的深度优化。

选择数据库开发工作主要是为了能够结合计算机理论知识，实践分布式、存储、数据库、内核，反过来在实践开发过程中巩固提高自己。

图灵访谈：阿里云的Redis数据库和其他云计算厂商相比，有何优点？

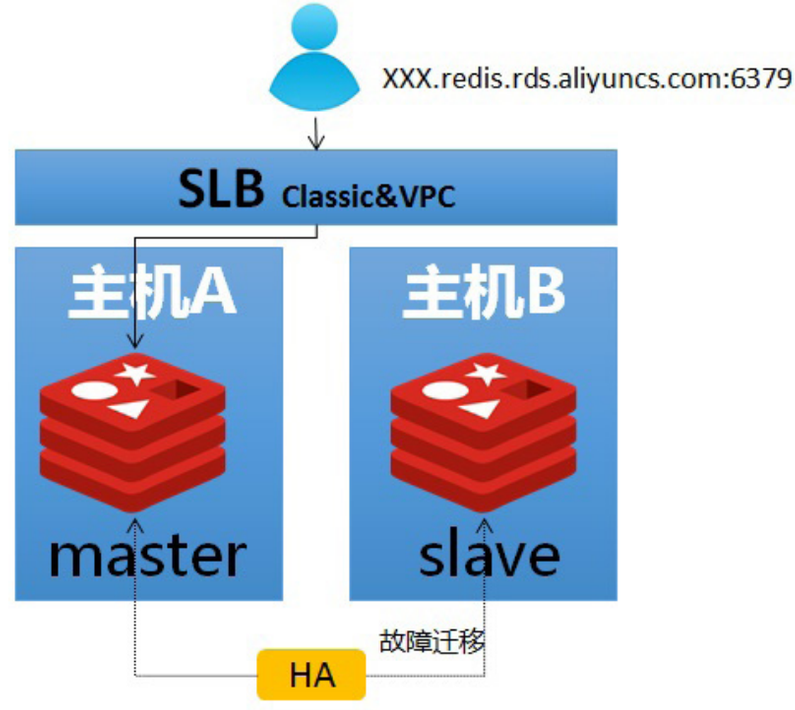
阿里云的Redis云服务整体架构在ApsaraDB上，具有完善的任务管理、监控运维、高可用体系，能够支撑海量的实例管理及不同的产品需求。阿里云Redis数据库提供了主从双副本、主从单副本、集群双副本等多样的产品供用户选择。



主从双副本

主从双副本版本要求主从双机热备，将数据持久化到磁盘，当主库发生故障的时候快速切换到备库上以保证服务的可用性。相比开源Redis

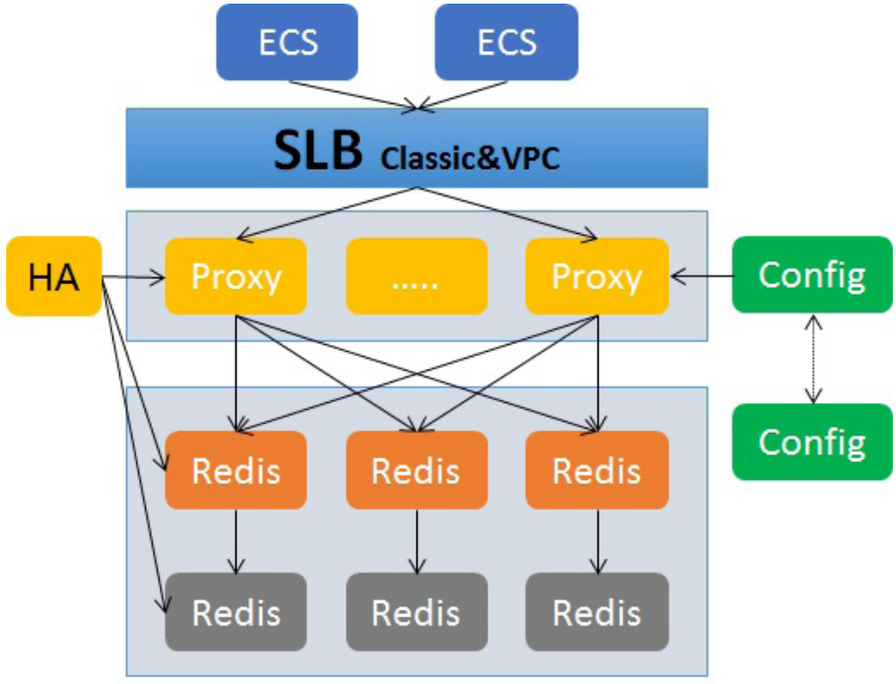
及其他厂商的Redis，阿里云对Redis的故障探测进行了更深度地优化：通过专门的探测端口来避免Redis单线程阻塞的影响，通过对磁盘、CPU、内存等硬件的检测提前发现故障隐患进行主动切换，优化原生的主备复制机制，采用增量日志加内存buffer的方式进行同步，避免弱网情况下主备复制频繁断开及全量同步。



主从单副本

主从单副本是云数据库 Redis 推出的一种全新系列，采用单个数据库节点部署架构。与双副本版本相比，它只包含一个节点，没有备用节点实时同步数据，不提供数据持久化和备份策略，适用于对数据可靠性要求不高的纯缓存业务场景。与其他云厂商相比，阿里云Redis单副本能够

保证在主库发生故障的时候快速切换到新节点，但是这个新节点是没有数据的，用户需要在切换完成之后进行数据预热避免数据库的打穿。



集群双副本

阿里云Redis集群完全采用自研的技术体系，设计的时候充分考虑到用户能够从主从版本平滑迁移过来的需求。Redis集群架构引入了Proxy、Config模块，Proxy负责数据的分发及路由，Config负责数据的迁移及路由表管理。多个无状态的Proxy、Config模块可以保证整个链路的高可用，避免单点问题。与其他云厂商相比，阿里云Redis集群能够保证更高的兼容性，用户可以在主从版本和Redis集群之间进行无缝切换，

不需要去更改用户的代码。同时，阿里云Redis集群还支持多db的模式，相比大部分开源的Redis集群方案有较大的优势。

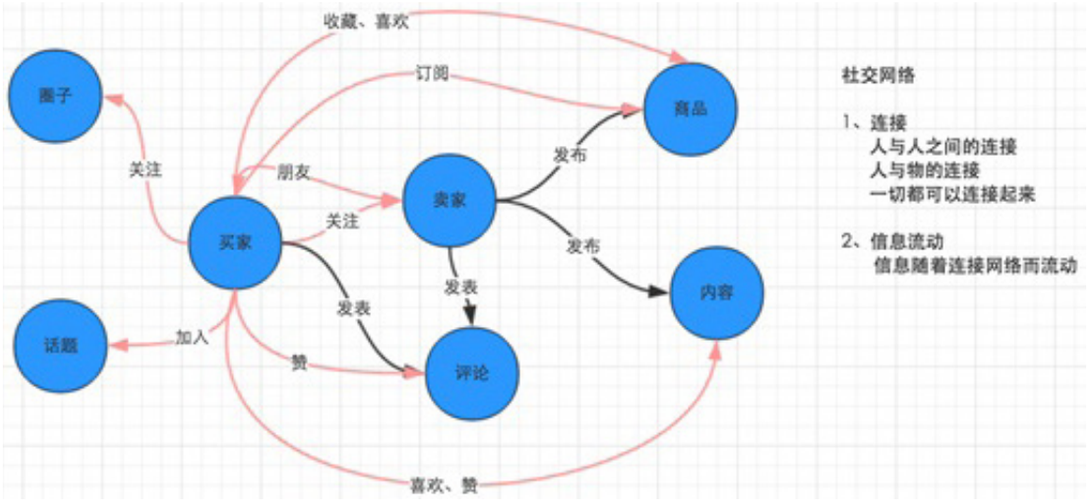
为了更好的服务用户，后续阿里云Redis产品还会推出读写分离、异地容灾、异地多活等产品形态供用户进行选择。最近阿里云Redis又推出了256MB的主从双副本实例，适用于PHP缓存、论坛加速、数据库加速等，随着业务的发展用户还可以进一步扩容。活动期间可以享受99元包年的优惠！

图灵访谈：先后接触过Tair，Memcached和Redis后，在云服务开发中有哪些深刻的体会？

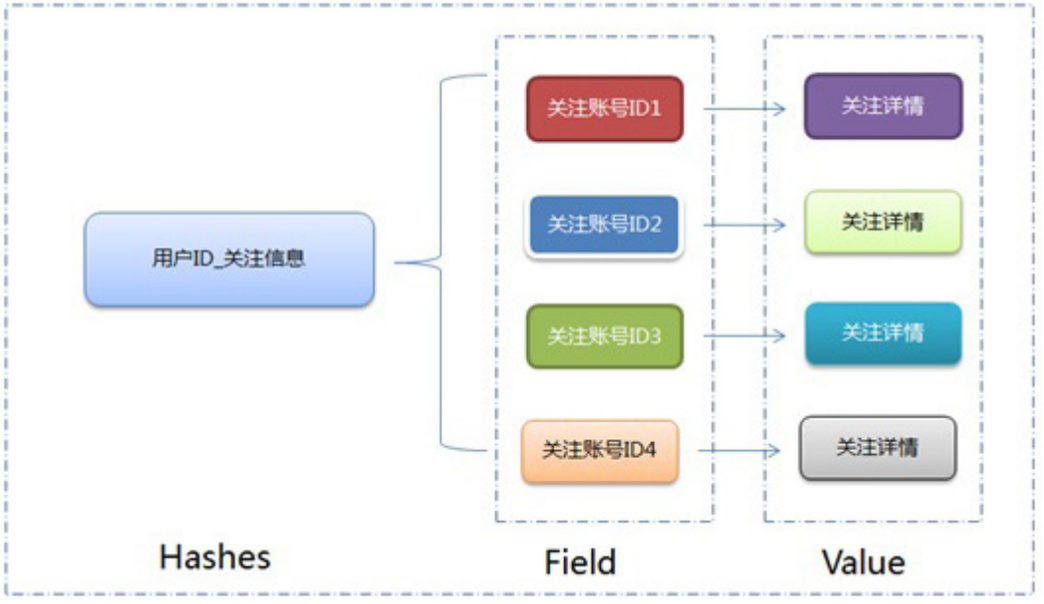
作为基础服务开发工程师，数据库的稳定性、性能是第一要位的。用户对数据库的高可用及稳定性都有很高要求的，我们在开发过程中需要选用合理的架构，在架构上避免单点问题，避免故障无法恢复问题。对于云服务需要做到足够的隔离，避免不同租户的互相影响，同时在设计云服务过程中需要考虑到可扩展性。公有云服务的每个业务都有自己的特点，会催生出不同的业务需求，所以在初步设计的时候需要保留支持多种业务模式的能力。

图灵访谈：在阿里云内部，Redis技术最得意的应用场景是什么？使用过程中存在哪些痛点？

阿里内部大量使用了阿里云Redis服务，比如手淘、高德、CDN业务、菜鸟等都大量使用了Redis以实现不同的业务。微淘社区承载了亿级淘宝用户的社交关系链，其中每个用户都有自己的关注列表，每个商家都有自己的粉丝信息。我用下面的图来展示整个微淘社区承载的关系链。



如果选用传统的关系型数据库模型表达上面的关系信息，业务设计会变得异常繁杂，也不能获得良好的性能体验。使用Redis集群，微淘社区缓存了存储社区的关注链，简化了关注信息的存储，并保证了双11业务丝滑一般的体验。微淘社区使用了Hashes存储用户之间的关注信息，存储结构如下图所示。



随着业务规模的壮大，用户需要后端Redis云服务能够做到动态扩容。阿里云Redis集群实例提供了资源变配功能，用户可以在需要的时候进行变配以应对容量的增加。另外，对于淘宝业务来说，每年的双11都是重中之重，我们在双11之前都会跟业务方确认当年的访问量和业务量，同时提前进行双11的全链路压测，保证业务丝滑般的体验。

在使用开源Redis的过程中，我们也碰到了很多问题，比如原生Redis的同步依赖于内存buffer，这会带来一个问题：在弱网情况下，如果内存buffer溢出，原生Redis需要进行一次全量同步。为此阿里云Redis对主备同步进行了优化，通过binlog日志加内存buffer的形式解决掉弱网全量同步的缺陷。另外，在云服务开发运维过程中难免需要对Redis服务进行升级管理，但原生的Redis内核不能很好地支持热升级机制，如果直接重启会对用户的访问产生很大的影响。阿里云Redis通过拆分动态库的形式做到了3ms内对一个实例进行热升级，而且升级过程中对用户的访问不会有任何影响。

图灵访谈：关系型和非关系型数据库在实现上有哪些主要区别？各自面向的应用领域和作用是什么？

简单来说，关系型数据库是指采用关系模型来组织的数据库。传统的关系型数据库由二维表模型来组织，表与表之间具有一定联系，业务可以通过SQL语句对数据库进行更新、查找、删除。非关系型数据库（NoSQL）的最初定义是没有SQL的轻量级数据库，并且不保证遵循ACID原则的数据存储系统。

关系型和非关系型最大的区别在于是否保持事务的一致性，虽然像Redis这样的非关系型数据库也有事务的说法，不过Redis事务是相对简单的事务模型，而传统关系型的数据库是要求读写操作来保证事务一致性的，因为关系型的数据库具有更多的应用场景，并且多用于对数据一致性有强烈要求的系统中。非关系型数据库里的key—value结构具有极高的并发读写性能，所以常常用于高速缓存中，作为传统数据库的缓存提供更高的并发访问。

图灵访谈：新手如何选择数据库类型，传统关系型数据库还是非关系型数据库？

两种数据库类型是相辅相成的，关系型数据库可以用于对数据一致性有更高要求的场景，同时能够支撑复杂类型的关联查询，而非关系型的数据库可以用于数据库的缓存，或者用于结构简单的业务场景。另外，由于Redis提供了更多复杂类型的数据结构，所以也可以将Redis用于更丰富的业务场景，比如用List结构来实现推送系统、弹幕系统等。

图灵访谈：数据库的实现相当复杂，如果想要研究如何实现数据库系统，即学习造轮子，您有什么建议？

如果要研究数据库的实现可以先从应用层的应用开始，先熟悉数据库的应用场景，也可以先阅读数据库相关的基础知识，了解数据库的SQL解析、持久化机制、数据同步、数据库索引等掌握数据库实现的设计原理。学习过程中，我们可以阅读优秀开源数据库的源码实现，跟踪调试了解整个数据库命令执行的过程；根据数据库命令的每一步执行，在代码跟进过程中思考如何优化和增加新功能。积极融入开源社区的各种活动，定

期查看社区的问题。在解决社区用户汇报问题的过程中，对数据库知识做到整体地把握，着重深入某一具体方面的知识。

以NOSQL数据库的学习为例，我们可以通过阅读Redis数据库源码，了解每种数据结构在内存里的组织方式，思考如何更好地存储复杂类型的数据结构。跟Memcached进行比较，分析两种做法的优劣。

为了更好地研究数据库的实现，我们需要多动手，在实际项目中了解数据库的实现，通过开发新功能、优化老功能来巩固知识，同时兼顾数据库理论知识的学习。

图灵访谈：如何保证缓存数据和数据库数据的一致性？

在大数据、高并发的情况下，很多业务系统都会采用“缓存+数据库”的方式，对缓存副本和数据库数据做同步处理。同步的方式主要有以下几种。

- 自动失效：通过设置过期时间让缓存中的数据自动失效，对数据一致性要求更高的业务，可以将过期时间设置得短一点
- 定时刷新：通过后台设置定时刷新的线程，定时将数据库的数据同步到缓存中，这种模式适合对一致性要求不是很高的业务
- 同步更新：在更新完成数据库之后，直接更新缓存的数据。这样缓存的数据就永远在数据库之后更新，能够保证数据的一致性
- 中间件更新：通过中间件订阅数据库的binlog服务，感知数据库信息的变化在中间件上对缓存进行更新，这种模式能够快速感知数据库的变化并且不需要业务方去管理缓存，方便业务接入缓存系统。

图灵访谈：谈谈分布式系统存储技术未来的方向？

由于传统单机数据库在可扩展性上面临着巨大的挑战，而NoSQL并不能很好地支持关系型的数据库模型，NewSQL成为了目前分布式数据库存储技术最前沿的一个方向。NewSQL具有海量数据的存储管理并且能够保持传统数据库的ACID及SQL特性。

NewSQL采用了大量的新技术。在存储方面，NewSQL采用以内存为主的存储，将主要的数据缓存于内存中，能够维持内存和持久化存储的数据比例，提高系统的性能；为了支持数据库的可扩展性，NewSQL通过数据分片的模式将数据分布到不同的group中，同时能够支持不同group的数据进行热迁移以达到数据的负载均衡；NewSQL在复制方面需要保证数据的一致性，事务的写入必须在被确认提交之前被确认并安装到所有副本；要做到高可用就需要从崩溃中恢复，相比传统的故障恢复，NewSQL还希望尽量减少故障恢复的时间。

从技术角度来讲，NewSQL与传统数据库并不是完全不同的架构，NewSQL的新特点在于能够融合多方的技术，在一个独立的系统中进行实现，同时随着硬件技术的发展能够提供更可靠更具有扩展性的数据库服务。■

——更多访谈

用正确的方式，打开Go测试

译者 /dogstar

在学习任何新事物的时候，具备清醒的头脑很重要。

如果你对Go相当陌生，并且来自诸如JavaScript或Ruby这样的语言，你很可能习惯于使用现成的框架来模拟、断言以及做一些其他的测试。

不过，从现在开始，消除依赖外部或框架的想法！几年前，我在学习Go这门出众的编程语言时，遇到了第一个障碍——测试，也许是当时的可用资源相当少吧。

现在我知道了，如果想在GO中测试成功，就意味着减少对外部类库的依赖，编写更好、可重用的代码。

Go中的表格测试

基本的测试单元——“单元测试”——可以是一个程序的任何部分，只需要一个输入并返回一个输出。下面，我们来看一个简单的函数，之后为它编写测试。显然，该函数并不完美也不完整，但作为演示，足够了：

avg.go

```
func Avg(nos ...int) int {
    sum := 0
```

```
    for _, n := range nos {
        sum += n
    }
    if sum == 0 {
        return 0
    }
    return sum / len(nos)
}
```

func Avg(nos ...int)函数返回零，或者返回一系列数字的整数平均值。现在，我们来给它编写一个测试。

在Go语言中，测试文件的命名和包含待测试代码的文件名称相同，同时附加后缀_test。例如，上面的代码存在于名为avg.go的文件当中，所以我们将测试文件命名为avg_test.go。

注意，这些示例只是实际文件的部分摘录，其中的包定义和导入已经删去。

这是针对Avg函数的测试：

avg_test.go

```
func TestAvg(t *testing.T) {
    for _, tt := range []struct {
        Nos    []int
        Result int
    }{
        {Nos: []int{2, 4}, Result: 3},
        {Nos: []int{1, 2, 5}, Result: 2},
        {Nos: []int{1}, Result: 1},
        {Nos: []int{}, Result: 0},
        {Nos: []int{2, -2}, Result: 0},
    } {
        if avg := Average(tt.Nos...); avg != tt.Result {
```

```
                t.Fatalf("expected average of %v to be %d, got %d\n", tt.Nos,
tt.Result, avg)
        }
    }
}
```

关于函数的定义，这里有几件事情需要注意：

- 首先，测试函数名称中的Test前缀必需保留，以便工具把它作为一种有效的测试检测出来。
- 测试函数名称的后半部分通常是待测试函数或者方法的名称，在这里是Avg。
- 还需要传入testing.T的测试结构，允许控制测试流。有关此API的更多详细信息，请访问此文档。

现在，我们来聊聊这个例子的编写格式。测试套件（一系列的测试）正通过Agv()函数运行，并且每个测试包含一个特定的输入和预期的输出[1]。在我们的例子中，每次测试会传入一系列整数（Nos）和所期望的一个特定的返回值（Result）。

[1] 表格测试从结构得名，所以很容易被理解成一个包含两列变量的表格：输入变量和预期的输出变量。

Golang接口模拟

Go语言所提供的最伟大也是最强大的功能就是接口[2]。除了程序架构设计时获得接口的强大功能和灵活性以外，接口也提供了令人惊讶的机会帮助我们解耦组件并在交汇点全面测试组件。

[2] 接口是指定方法的集合，也是一个变量类型。

我们看一个虚构的场景：需要从io.Reader读取前*N*个字节，并把它们作为一个字符串返回。

readn.go

```
// readN从r中最多读取N个字节，并把它们作为字符串返回
func readN(r io.Reader, n int) (string, error) {
    buf := make([]byte, n)
    m, err := r.Read(buf)
    if err != nil {
        return "", err
    }
    return string(buf[:m]), nil
}
```

显然，主要测试readN这个功能，即给定各种输入时，返回正确的输出。我们可以用表格测试来完成，但需要覆盖到两个特殊的场景：

- readN被一个大小为n的缓冲调用
- 如果抛出异常，readN返回错误

为了知道传递给r.Read的缓冲区的大小，以及控制它返回的错误，我们需要模拟传递给readN的r。如果看一下Go文档中的Reader类型，io.Reader看起来像：

```
type Reader interface {
       Read(p []byte) (n int, err error)
}
```

这似乎相当容易。为了满足io.Reader，我们需要模拟一个Read方法。所以，ReaderMock可以是这样：

```
type ReaderMock struct {
    ReadMock func([]byte) (int, error)
```

```
}

func (m ReaderMock) Read(p []byte) (int, error) {
    return m.ReadMock(p)
}
```

稍微分析一下上面的代码。ReaderMock 明显满足了 io.Reader 接口，因为它实现了必要的 Read 方法。我们的模拟还包含字段 ReadMock，用于设置模拟方法确切的行为。

为确保接口在运行时能满足需要，一个伟大而不消耗内存的技巧是，把以下的代码插入到我们的代码中：

```
var _ io.Reader = (*MockReader)(nil)
```

这样利于检查断言，但不会分配任何东西。我们可以确保该接口在编译时已被正确实现。可选的技巧，但很实用。

继续往前，我们来写第一个测试：r.Read 被大小为 n 的缓冲区调用。为了做到这点，我们运用 ReaderMock：

```
func TestReadN_bufSize(t *testing.T) {
    total := 0
    mr := &MockReader{func(b []byte) (int, error) {
        total = len(b)
        return 0, nil
    }}
    readN(mr, 5)
    if total != 5 {
        t.Fatalf("expected 5, got %d", total)
    }
}
```

正如你看到的，我们通过一个局部变量定义了“假的”io.Reader功能，用于后面断言测试的有效性。相当容易！

再来看下需要测试的第二个场景，我们需要模拟Read返回错误：

```
func TestReadN_error(t *testing.T) {
    expect := errors.New("some non-nil error")
    mr := &MockReader{func(b []byte) (int, error) {
        return 0, expect
    }}
    _, err := readN(mr, 5)
    if err != expect {
        t.Fatal("expected error")
    }
}
```

在上面的测试中，不管mr.Read（我们模拟的Reader）被谁调用，都会返回错误。因此，假设readN的正常运行是可靠的。

Golang方法模拟

通常情况下，我们不需要模拟方法，使用结构和接口就可以。偶尔会碰到必须模拟方法的时候，我也经常遇到这方面的困惑，甚至有人问我怎么模拟类似log.Println的东西。我会利用这次机会向大家证明很少测试log.Println的输入。

考虑以下的if语句，根据n的值输出记录：

```
func printSize(n int) {
    if n < 10 {
```

```
        log.Println("SMALL")
    } else {
        log.Println("LARGE")
    }
}
```

在上面的例子中，我们假设了这样一个可笑的场景：特定测试log.Println被正确的值调用。为了模拟这个功能，首先需要把它包装起来：

```
var show = func(v ...interface{}) {
    log.Println(v...)
}
```

间接地把log.Println的代码行替换成show，程序会变成：

```
func printSize(n int) {
    if n < 10 {
        show("SMALL")
    } else {
        show("LARGE")
    }
}
```

现在我们可以测试了：

```
func TestPrintSize(t *testing.T) {
    var got string
    oldShow := show
    show = func(v ...interface{}) {
        if len(v) != 1 {
            t.Fatalf("expected show to be called with 1 param, got %d", len(v))
        }
        var ok bool
        got, ok = v[0].(string)
        if !ok {
```

```
            t.Fatal("expected show to be called with a string")
        }
    }

    for _, tt := range []struct{
        N int
        Out string
    }{
        {2, "SMALL"},
        {3, "SMALL"},
        {9, "SMALL"},
        {10, "LARGE"},
        {11, "LARGE"},
        {100, "LARGE"},
    } {
        got = ""
        printSize(tt.N)
        if got != tt.Out {
            t.Fatalf("on %d, expected '%s', got '%s'\n", tt.N, tt.Out, got)
        }
    }

    // 虽然很小心，我们一定不要忘记在完成测试之前恢复初始值
    // 否则，它会跟其他的测试互相干扰，给行为追踪产生无法预计的困难
    show = oldShow
}
```

我们不应该“模拟log.Println”，但在非常偶然的情况下，真的需要模拟一个包级的方法时，唯一的方法（据我所知）就是把它声明为一个包级的变量，这样我们就可以控制它的值了。

如果确实需要模拟像log.Println这样的东西，又使用了自定义的记录器，我们可以编写一个更优雅的解决方案。

Golang 模板渲染测试

另一个相当常见的情况是，根据预期测试某个渲染模板的输出。我们考虑一个对 http://localhost:3999/welcome?name=Frank 的 GET 请求，它会返回以下 body：

```
<html>
    <head><title>Welcome page</title></head>
    <body>
        <h1 class="header-name">
            Welcome <span class="name">Frank</span>!
        </h1>
    </body>
</html>
```

❸ GoQuery 使用类似 jQuery 的 API 查询 HTML 结构，是测试程序标签输出有效性所必不可少的类库。

如果现在还不够明显，查询参数 name 与类为 name 的 span 标签相匹配，这不是一个巧合。在这种情况下，明显的测试应该验证每次跨越多层输出时这种情况都正确发生。在这里我发现 GoQuery 类库 ❸ 非常有用。

用这种方式我们现在可以编写测试了：

```
welcome__test.go

func TestWelcome_name(t *testing.T) {
    resp, err := http.Get("http://localhost:3999/welcome?name=Frank")
    if err != nil {
        t.Fatal(err)
    }
    if resp.StatusCode != http.StatusOK {
        t.Fatalf("expected 200, got %d", resp.StatusCode)
    }
    doc, err := goquery.NewDocumentFromResponse(resp)
```

```
    if err != nil {
        t.Fatal(err)
    }
    if v := doc.Find("h1.header-name span.name").Text(); v != "Frank" {
        t.Fatalf("expected markup to contain 'Frank', got '%s'", v)
    }
}
```

在处理前，我们检查响应状态码是不是200/OK。

我认为，假设上面代码段的其余部分是不言自明的，这并不会太牵强：我们使用`http`包来提取URL并根据响应创建一个新的`goquery`兼容文档，随后我们会用它来查询返回的DOM。我们测试了`h1.header-name`里面`span.name`的封装文本“弗兰克”。

测试JSON接口

Golang经常用来写某种API，所以，在最后的部分，我们会看一些测试JSON API高级的方式。

试想，如果前面的终端返回的是JSON，而不是HTML，我们从`http://localhost:3999/welcome.json?name=Frank`得到的响应body看起来会是这样：

```
{"Salutation": "Hello Frank!"}
```

正如你能想到那样，断言JSON响应相比于断言模板响应来说并不会有太大的不同，有一点特殊就是我们不需要任何外部库或依赖。仅用Golang的标准库就足够了。下面是我们的测试，以确认对于给定的参数返回正确的JSON：

welcome__test.go

```
func TestWelcome_name_JSON(t *testing.T) {
    resp, err := http.Get("http://localhost:3999/welcome.json?name=Frank")
    if err != nil {
        t.Fatal(err)
    }
    if resp.StatusCode != 200 {
        t.Fatalf("expected 200, got %d", resp.StatusCode)
    }
    var dst struct{ Salutation string }
    if err := json.NewDecoder(resp.Body).Decode(&dst); err != nil {
        t.Fatal(err)
    }
    if dst.Salutation != "Hello Frank!" {
        t.Fatalf("expected 'Hello Frank!', got '%s'", dst.Salutation)
    }
}
```

考虑到对响应结构成功解码后，我们检查字段的内容是否达到预期的情况，即“Hello Frank!”。如果返回了解码以外的任何结构，`json.NewDecoder`反而会返回一个错误，测试失败。

Setup和Teardown

Golang的测试是容易的，但有一个问题是，在这之前的JSON测试和模板渲染测试都假设服务器正在运行，而这创建了一个不可靠的依赖。另外，需要一个“活”的服务器并不是一个很好的主意4！

在一个“活”的生产服务器上对“实时”数据进行测试从来都不是一个好主意。

幸运的是，Golang 提供了 httptest 包来创建测试服务器。如果测试引发了独立于主要服务器以外的服务器，测试就不会干扰生产环境。

这种情况下，创建通用的 setup 和 teardown 方法以便被需要运行服务器的全部测试调用是理想的方法。根据这种模式，最终的测试看起来会像是这样：

```
func setup() *httptest.Server {
    return httptest.NewServer(app.Handler())
}

func teardown(s *httptest.Server) {
    s.Close()
}

func TestWelcome_name(t *testing.T) {
    srv := setup()

    url := fmt.Sprintf("%s/welcome.json?name=Frank", srv.URL)
    resp, err := http.Get(url)
    // 想往常一样，验证错误并进行断言

    teardown(srv)
}
```

注意 app.Handler() 的引用。这是一个最佳实践的函数，它返回了应用程序的 http Handler，这既可以实例化生产服务器也可以实例化测试服务器。

结论

Golang 测试是一个很好的机会，它假定了程序的外部视野，承担了访问者的脚步（在大多数情况下，就是 API 的用户），确保了良好的代码和优质的体验。

当不确定代码中的复杂功能时，测试作为一颗定心丸就会派上用场，它可以保证系统的其他部分面对较大改动时仍能继续运作。

希望这篇文章能对你有用。如果你知道其他任何的测试技巧，也欢迎来发表评论。

文章转自译者 dogstar 发表于艾翻译 (itran.cc) 的文章。

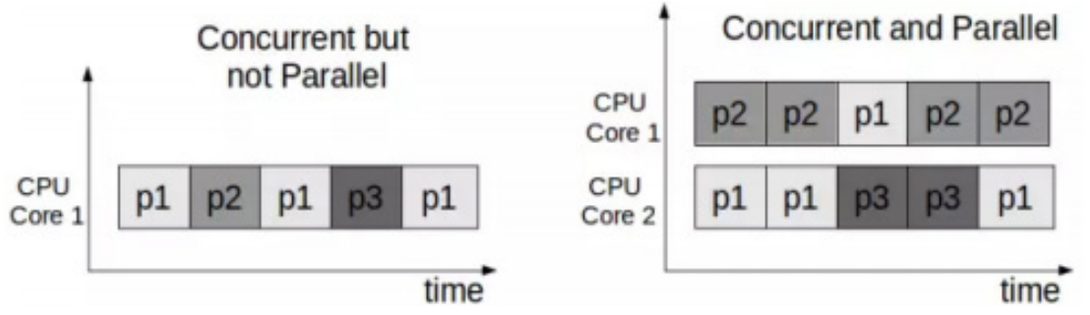
解决并发之痛

作者 / 王渊命

网名 Jolestar，曾任新浪微博架构师，微米技术总监。当前在青云负责容器平台的相关开发工作。关注大数据、分布式系统、容器与云。各种语言都有涉猎，主要使用 Java 和 Go。

个人技术博客：http://jolestar.com

聊并发之痛这个话题之前，我先梳理下两个概念，几乎所有讲并发的文章也都会先讲这两个概念：



- **并发（concurrency）** 并发的关注点在任务的切分。举例来说，如果你是一个创业公司的 CEO，并且公司开始只有你一个人，所以你必须一人分饰多角。一会做产品规划，一会写代码，一会见客户，虽然你不能见客户的同时写代码，但由于你切分了任务，分配了时间片，表现出来好像是多个任务一起在执行。
- **并行（parallelism）** 并行的关注点在同时执行。还是上面的例子，你发现自己太忙了，时间分配不过来，于是请了工程师、产品经理、市场总监，让他们各司一职，这时候多个任务就可以同时执行了。

所以总结下，并发并不要求必须并行。我们可以用时间片切分的方式模拟，比如单核cpu上的多任务系统，并发的要求是任务能切分成独立执行的片段。而并行关注的是同时执行，必须是多（核）cpu，需要能并行的程序必须是支持并发的。本文大多数情况下不会严格区分这两个概念，默认并发就是指并行机制下的并发。

为什么并发程序这么难？

> We believe that writing correct concurrent, fault-tolerant and scalable applications is too hard. Most of the time it ' s because we are using the wrong tools and the wrong level of abstraction.
>
> —— Akka

Akka官方文档开篇这句话说得好，写出正确的、可容错的、可扩展的并发程序之所以如此之难，是因为我们用了错误的工具和错误的抽象。（当然该文档本来的意思是Akka是正确的工具，但我们可以独立的看待这句话）。

我们从最开始梳理下程序的抽象。程序开始是面向过程的，数据结构+func。后来有了面向对象，对象组合了数结构和func，我们想用模拟现实世界的方式抽象出对象、有状态和行为。但无论是面向过程的func还是面向对象的func，本质上都是代码块的组织单元，本身并没有包含代码块的并发策略的定义。于是为了解决并发的需求，引入了Thread（线程）概念。

线程 (Thread)

- 系统内核态，更轻量的进程
- 由系统内核进行调度
- 同一进程的多个线程可共享资源

线程的出现解决了两个问题，一个是GUI出现后急切需要并发机制来保证用户界面的响应。第二是互联网发展后带来的多用户问题。最早的CGI程序很简单，通过脚本将原来单机版的程序包装在一个进程里，来一个用户就启动一个进程。但明显这样承载不了多少用户，并且如果进程间需要共享资源的话还得通过进程间的通信机制，线程的出现缓解了这个问题。

线程的使用比较简单，如果你觉得这块代码需要并发，就把它放在单独的线程里执行，由系统负责调度，具体什么时候使用线程，要用多少个线程，由调用方决定。但定义方并不清楚调用方会如何使用自己的代码，很多并发问题都是因为误用导致的，比如Go中的`map`以及Java的`HashMap`都不是并发安全的，误用在多线程环境就会导致问题。另外这也带来复杂度：

1. 竞态条件（race conditions）如果每个任务都是独立的，不需要共享任何资源，那线程也就非常简单。但世界往往是复杂的，总有一些资源需要共享，比如前面的例子，开发人员和市场人员同时需要和CEO商量一个方案，这时候CEO就成了竞态条件。

2. 依赖关系以及执行顺序 如果线程之间的任务有依赖关系，需要等待以及通知机制来进行协调。比如前面的例子，如果产品和CEO讨论的方案依赖于市场和CEO讨论的方案，这时候就需要协调机制保证顺序。

为了解决上述问题，我们引入了许多复杂机制来保证：

- Mutex(Lock)（Go里的sync包，Java的concurrent包）通过互斥量来保护数据，但有了锁，明显就降低了并发度。
- semaphore 通过信号量来控制并发度或者作为线程间信号（signal）通知。
- volatile Java专门引入了volatile关键词，来降低只读情况下的锁的使用。
- compare-and-swap 通过硬件提供的CAS机制保证原子性（atomic），也是降低锁的成本的机制。

如果说上面两个问题只是增加了复杂度，我们通过深入学习，严谨地CodeReview，全面的并发测试（比如Go语言中单元测试的时候加上-race参数），一定程度上能解决。（当然这个也是有争议的，有论文认为当前的大多数并发程序没出问题只是并发度不够，如果CPU核数继续增加，程序运行的时间更长，很难保证不出问题）但是，最让人头痛的还是下面这个问题：

系统里到底需要多少线程？

针对这个问题我们先从硬件资源入手，考虑下线程的成本：

- 内存（线程的栈空间）
  每个线程都需要一个栈（Stack）空间来保存挂起（suspending）时的状态。Java 的栈空间（64 位 VM）默认是 1024k。不算别的内存，只是栈空间，启动 1024 个线程就要 1G 内存。虽然可以用 -Xss 参数控制，但由于线程本质上也是进程，系统假定是要长期运行的，栈空间太小会导致稍复杂的递归调用（比如复杂点的正则表达式匹配）以及导致栈溢出。所以调整参数治标不治本。

- 调度成本（context-switch）
  我在个人电脑上做过一个非严格的测试，模拟两个线程互相唤醒轮流挂起。线程切换成本大约 6000 纳秒 / 次，这还没考虑栈空间大小的影响。国外一篇论文专门分析线程切换的成本，基本上得出的结论是切换成本和栈空间使用大小直接相关。

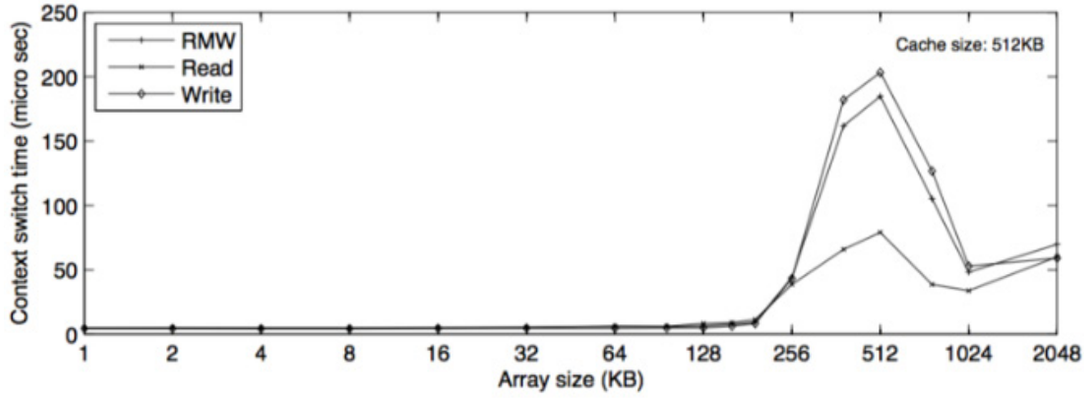


Figure 1: The effect of data size on the cost of the context switch

- CPU使用率

  我们搞并发最主要的一个目标，就是有了多核进而想提高CPU的利用率，最大限度地压榨硬件资源。从这个角度考虑，我们应该用多少线程呢？

100毫秒的时间片，该线程实际使用运算15毫秒，等待网络80毫秒，解析和处理结果返回5毫秒，我们有4核CPU，应该多少线程合适？

我们可以通过一个公式计算出来，100/(15+5)*4=20，用20个线程最合适。但一方面网络的时间不是固定的，另外一方面，如果考虑到其他瓶颈资源呢？比如锁，比如数据库连接池，就会更复杂。

作为一个1岁多孩子的父亲，我认为这个问题的难度好比要写个给孩子喂饭的程序，需要考虑“给孩子喂多少饭合适？”。这个问题有以下回答以及策略：

- 孩子不吃了就好了（但孩子贪玩，不吃了可能是想去玩了）
- 孩子吃饱了就好了（废话，你怎么知道孩子吃饱了？孩子又不会说话）
- 逐渐增量，长期观察，然后计算一个平均值（这可能是我们调整线程常用的策略，但增量增加到多少合适呢？）

- 孩子吃吐了就别喂了（如果用逐渐增量的模式，通过外部观察，可能会到达这个边界条件。系统性能如果因为线程的增加倒退了，就别增加线程了）
- 没控制好边界，把孩子给给撑坏了（这熊爸爸也太恐怖了！但调整线程的时候往往不小心可能就把系统搞挂了）

通过这个例子我们可以看出，从外部系统来观察，或者以经验的方式进行计算，都是非常困难的。于是结论是：**让孩子会说话，吃饱了自己说，自己学会吃饭，自管理。**

然而并没有什么作用，计算机不会自己说话，如何自管理?

但我们从以上的讨论中可以得出一个结论：

- 线程的成本较高（内存，调度）不可能大规模创建
- 应该由语言或者框架动态解决这个问题

线程池方案

Java 1.5后，Doug Lea的Executor系列被包含在默认的JDK内，这是典型的线程池方案。

线程池一定程度上控制了线程的数量，实现了线程复用，降低了线程的使用成本。但还是没有解决数量的问题，线程池初始化的时候还是要设置一个最小和最大线程数，以及任务队列的长度，自管理只是在设定范围内的动态调整。另外，不同的任务可能有不同的并发需求，为了避免

互相影响我们可能需要多个线程池，最后导致的结果就是Java的系统里充斥了大量的线程池。

新思路

从前面的分析我们可以看出，如果线程是一直处于运行状态，我们只需设置和CPU核数相等的线程数，这样就可以最大化地利用CPU，并且降低切换成本以及内存使用。但如何做到这一点呢?

陈力就列，不能者止

这句话是说，能干活的代码片段就放在线程里，如果干不了活（需要等待，被阻塞等），就摘下来。通俗地说就是不要占着茅坑不拉屎，如果拉不出来需要酝酿下，先把茅坑让出来，因为茅坑是稀缺资源。

要做到这点一般有两种方案:

1. **异步回调方案 典型如NodeJS**，遇到阻塞的情况，比如网络调用，则注册一个回调方法（其实还包括了一些上下文数据对象）给IO调度器（linux下是libev，调度器在另外的线程里），当前线程就被释放了，去干别的事情了。等数据准备好，调度器会将结果传递给回调方法然后执行，执行其实不在原来发起请求的线程里了，但对用户来说并无感知。但这种方式的问题就是很容易遇到callback hell，因为所有的阻塞操作都必须异步，否则系统就卡死了。还有就是异步的方式有点违反人类思维习惯，人类还是习惯同步的方式。

2.GreenThread/Coroutine/Fiber 方案 这种方案其实和上面的方案本质上区别不大，关键在于回调上下文的保存以及执行机制。为了解决回调方法带来的难题，这种方案的思路是，写代码的时候还是按顺序写，但遇到IO等阻塞调用时，将当前的代码片段暂停，保存上下文，让出当前线程。等IO事件回来，再找个线程让当前代码片段恢复上下文继续执行，写代码的时候感觉好像是同步的，仿佛在同一个线程完成的，但实际上系统可能切换了线程，但对程序无感。

GreenThread

- 首先是在用户空间，避免由于内核态和用户态的切换增加成本
- 由语言或者框架层调度
- 更小的栈空间允许创建大量实例（百万级别）

这里提前铺垫几个概念。

- Continuation 如果不熟悉FP编程的人，可能不太熟悉。不过，这里可以简单地做下顾名思义，可以理解为让程序可以暂停，然后下次调用继续（contine）从上次暂停的地方开始的一种机制。相当于程序调用多了一种入口。
- Coroutine 它是Continuation的一种实现，一般表现为语言层面的组件或者类库。主要提供`yield`、`resume`机制。
- Fiber 它和Coroutine其实是一体两面的，主要是从系统层面描述，可以把Coroutine运行之后的东西理解成Fiber。

Goroutine

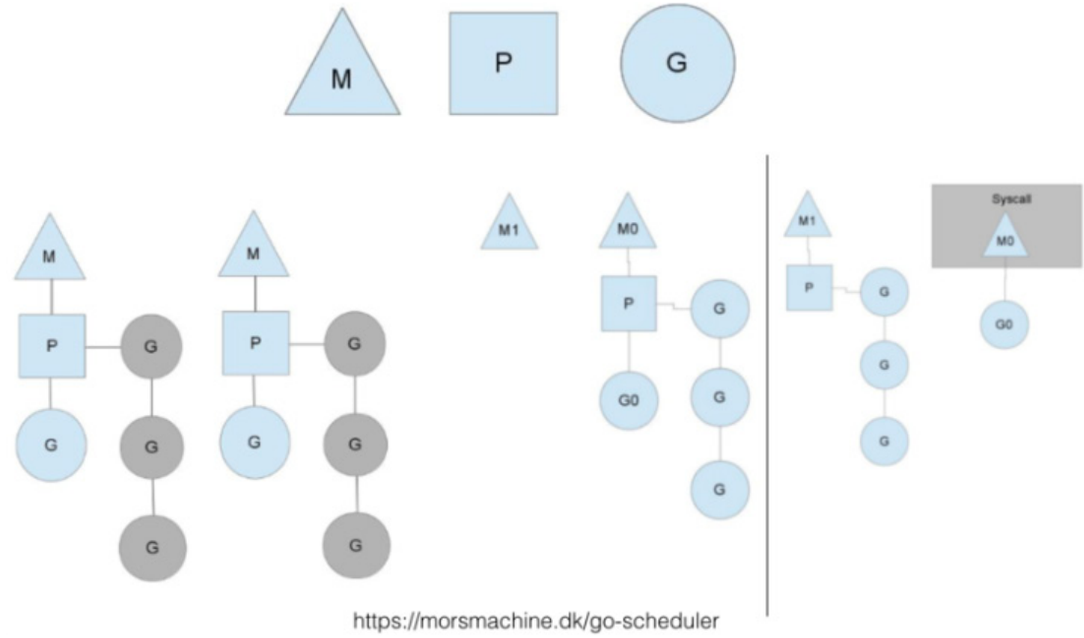
Goroutine 其实就是前面 GreenThread 系列解决方案的一种演进和实现。

首先，它内置了 Coroutine 机制。因为要用户态的调度，必须有可以让代码片段可以暂停 / 继续的机制。

其次，它内置了一个调度器，实现了 Coroutine 的多线程并行调度，同时通过对网络等库的封装，对用户屏蔽了调度细节。

最后，提供了 Channel 机制，用于 Goroutine 之间通信，实现 CSP 并发模 型 (Communicating Sequential Processes)。因 为 Go 的 Channel 是通过语法关键词提供的，对用户屏蔽了许多细节。其实 Go 的 Channel 和 Java 中的 SynchronousQueue 是一样的机制，如果有 buffer 其实就是 ArrayBlockQueue。

Goroutine 调度器



这个图在讲Goroutine调度器的地方一般都会引用，想要仔细了解的话，可以看看原博客。这里只说明几点：

1. M代表系统线程，P代表处理器（核），G代表Goroutine。Go实现了M:N的调度，也就是说线程和Goroutine之间是多对多的关系。这点在许多GreenThread/Coroutine的调度器并没有实现。比如Java1.1版本之前的线程其实是GreenThread（这个词就来源于Java），但由于没实现多对多的调度，也就是没有真正实现并行，发挥不了多核的优势，所以后来改成基于系统内核的Thread实现了。

2. 某个系统线程如果被阻塞，排列在该线程上的Goroutine会被迁移。当然还有其他机制，比如M空闲了，如果全局队列没有任务，可能会从其他M偷任务执行，相当于一种rebalance机制。这里不再细说，有需要看专门的分析文章。

3. 具体的实现策略和我们前面分析的机制类似。系统启动时，会启动一个独立的后台线程（不在Goroutine的调度线程池里），启动netpoll的轮询。当有Goroutine发起网络请求时，网络库会将fd（文件描述符）和pollDesc（用于描述netpoll的结构体，包含因为读/写这个fd而阻塞的Goroutine）关联起来，然后调用runtime.gopark方法，挂起当前的Goroutine。当后台的netpoll轮询获取到epoll（linux环境下）的event，会将event中的pollDesc取出来，找到关联的阻塞Goroutine，并进行恢复。

Goroutine是银弹么？

Goroutine很大程度上降低了并发的开发成本，是不是我们所有需要并发的地方直接go func就搞定了呢？

Go通过Goroutine的调度解决了CPU利用率的问题。但遇到其他的瓶颈资源如何处理？比如带锁的共享资源，比如数据库连接等。互联网在线应用场景下，如果每个请求都扔到一个Goroutine里，当资源出现瓶颈的时候，会导致大量的Goroutine阻塞，最后用户请求超时。这时候就需要用Goroutine池来进行控流，同时问题又来了：池子里设置多少个Goroutine合适？

所以这个问题还是没有从更本上解决。

Actor模型

Actor对没接触过这个概念的人可能不太好理解，Actor的概念其实和OO里的对象类似，是一种抽象。面对对象编程对现实的抽象是**对象=属性+行为**（method），但当使用方调用对象行为（method）的时候，其实占用的是调用方的CPU时间片，是否并发也是由调用方决定的。这个抽象其实和现实世界是有差异的。现实世界更像Actor的抽象，互相都是通过异步消息通信的。比如你对一个美女say hi，美女是否回应以及如何回应是由美女自己决定的，并运行在美女自己的大脑里，并不会占用发送者的大脑。

所以Actor具有以下特征：- 计算 Actor可以做计算的，不需要占用调用方的CPU时间片，并发策略也是由自己决定。 - 存储 Actor可以保存状态 - 通信 Actor之间可以通过发送消息通讯

Actor遵循以下规则：

- 发送消息给其他的Actor
- 创建其他的Actor
- 接受并处理消息，修改自己的状态

Actor的目标：

- Actor可独立更新，实现热升级。因为Actor互相之间没有直接的耦合，是相对独立的实体，可能实现热升级。
- 无缝弥合本地和远程调用。因为Actor使用基于消息的通讯机制，无论是和本地的Actor，还是远程Actor交互，都是通过消息，这样就弥合了本地和远程的差异。
- 容错。Actor之间的通信是异步的，发送方只管发送，不关心超时以及错误，这些都由框架层和独立的错误处理机制接管。
- 易扩展，天然分布式。因为Actor的通信机制弥合了本地和远程调用，本地Actor处理不过来的时候，可以在远程节点上启动Actor然后转发消息过去。

Actor的实现：

- Erlang/OTP Actor模型的标杆，其他的实现基本上都一定程度参照了Erlang的模式。实现了热升级以及分布式。
- Akka（Scala,Java）基于线程和异步回调模式实现。由于Java中没

有 Fiber，所以是基于线程的。为了避免线程被阻塞，Akka 中所有的阻塞操作都需要异步化。要么是 Akka 提供的异步框架，要么通过 Future-callback 机制，转换成回调模式。实现了分布式，但还不支持热升级。

- Quasar (Java) 为了解决 Akka 的阻塞回调问题，Quasar 通过字节码增强的方式，在 Java 中实现了 Coroutine/Fiber。同时通过 ClassLoader 的机制实现了热升级。缺点是系统启动的时候要通过 javaagent 机制进行字节码增强。

Golang CSP VS Actor

二者的格言都是：Don't communicate by sharing memory, share memory by communicating

通过消息通信的机制来避免竞态条件，但具体的抽象和实现上有些差异。

- CSP 模型里消息和 Channel 是主体，处理器是匿名的。
  也就是说，发送方需要关心自己的消息类型以及应该写到哪个 Channel，但不需要关心谁消费了它，以及有多少个消费者。Channel 一般都是类型绑定的，一个 Channel 只写同一种类型的消息，所以 CSP 需要支持 alt/select 机制，同时监听多个 Channel。Channel 是同步的模式（Golang 的 Channel 支持 buffer，支持一定数量的异步），背后的逻辑是发送方非常关心消息是否被处理，CSP 要保证每个消息都被正常处理了，没被处理就阻塞着。

- Actor模型里Actor是主体，Mailbox（类似于CSP的Channel）是透明的。

  也就是说，它假定发送方会关心消息发给谁消费了，但不关心消息类型以及通道。所以Mailbox是异步模式，发送者不能假定发送的消息一定被收到和处理。Actor模型必须支持强大的模式匹配机制，因为无论什么类型的消息都会通过同一个通道发送过来，需要通过模式匹配机制做分发。它背后的逻辑是现实世界本来就是异步的，不确定（non-deterministic）的，所以程序也要适应面对不确定的机制编程。自从有了并行之后，原来的确定编程思维模式已经受到了挑战，而Actor直接在模式中蕴含了这点。

从这样看来，CSP的模式比较适合Boss-Worker模式的任务分发机制，它的侵入性没那么强，可以在现有的系统中通过CSP解决某个具体的问题。它并不试图解决通信的超时容错问题，这个还是需要发起方进行处理。同时由于Channel是显式的，虽然可以通过netchan（原来Go提供的netchan机制由于过于复杂，被废弃，在讨论新的netchan）实现远程Channel，但很难做到对使用方透明。而Actor则是一种全新的抽象，使用Actor要面临整个应用架构机制和思维方式的变更。它试图要解决的问题要更广一些，比如容错，比如分布式。但Actor的问题在于以当前的调度效率，哪怕是用Goroutine这样的机制，也很难达到直接方法调用的效率。当前要像OO的『一切皆对象』一样实现一个『一切皆Actor』的语言，效率上肯定有问题。所以折中的方式是在OO的基础上，将系统的某个层面的组件抽象为Actor。

Rust

Rust解决并发问题的思路是，首先承认现实世界的资源总是有限的，想彻底避免资源共享是很难的，不试图完全避免资源共享，它认为并发的问题不在于资源共享，而在于错误地使用资源共享。比如，我们前面提到的，大多数语言定义类型的时候，并不能限制调用方如何使用，只能通过文档或者标记的方式（比如Java中的`@ThreadSafe`，`@NotThreadSafe annotation`）说明是否并发安全，但也只能仅仅做到提示的作用，不能阻止调用方误用。虽然Go提供了-race机制，可以通过运行单元测试的时候带上这个参数来检测竞态条件，但如果你的单元测试并发度不够，覆盖面不到也检测不出来。所以Rust的解决方案就是：

- 定义类型的时候，要明确指定该类型是否是并发安全的。
- 引入了变量的所有权（Ownership）概念。非并发安全的数据结构在多个线程间转移，也不一定就会导致问题，导致问题的是多个线程同时操作，也就是说是因为这个变量的所有权不明确导致的。有了所有权的概念后，变量只能由拥有所有权的作用域代码操作，而变量传递会导致所有权变更，从语言层面限制了竞态条件出现的情况。

有了这机制，Rust可以在编译期而不是运行期对竞态条件做检查和限制。虽然开发的时候增加了心智成本，但降低了调用方以及排查并发问题的心智成本，也是一种有特色的解决方案。

结论

革命尚未成功 同志任需努力

本文带大家一起回顾了并发的问题和各种解决方案。虽然各家有各家的优势以及使用场景，但并发带来的问题还远远没到解决的程度。所以还需努力，大家也有机会啊。

最后，抛个砖：构想在Goroutine上实现Actor？

- **分布式** 解决了单机效率问题，是不是可以尝试解决下分布式效率问题？
- **和容器集群融合** 当前的自动伸缩方案基本上都是通过监控服务器或者LoadBalancer，设置一个阀值来实现的。类似于我前面提到的喂饭的例子，是基于经验的方案，但如果系统内和外部集群结合，这个事情就可以做的更细致和智能。
- **自管理** 前面的两点最终的目标都是实现一个可以自管理的系统。做过系统运维的同学都知道，我们照顾系统就像照顾孩子一样，时刻要监控系统的各种状态，接受系统的各种报警，然后排查问题，进行紧急处理。孩子有长大的一天，那能不能让系统也自己成长，做到自管理呢？虽然这个目标现在看来还比较远，但我觉得是可以期待的。■

程序员到底能不能干过30岁？

作者/被搁浅的鱼

程序员为什么高薪？从经济学上来说是因为稀缺！但从世界范围来看，软件行业的从业者并不在少数。所以换个角度，程序员的薪资却有云泥之别，为什么？

回答这个问题之前，先谈一下前段时间使我对职业生涯感到恐慌的事件。华为被爆开除年龄大于34的员工，当时我下意识地算了算自己的年龄，不禁感慨如果自己就职华为，恐怕也快到了被开除的年纪了。刚工作时，我和许多不同专业背景的同学聊天，他们总会时不时地劝我，程序员是吃青春饭的，干不过30岁。虽然我们现在赚得少，可是稳定，你应该考虑转行之类的。说句实话，以前听到这些劝告时，心里多少还是有一些恐慌的，因为我实在想不出，除了写代码自己还能干些什么？

不是30岁的程序员不行了，而是你不行了

正如马云谈实体经济时的一句话，“不是实体经济不行了，而是你们家的实体经济不行了！”不是30岁的程序员不行了，而是你不行了。

软件行业之所以高薪，是因为这个行业的技术更新发展之快，远超其他行业。技术几乎每五年就翻新一轮，这就告诉行业里的每一位从业者一个事实：程序员的统一别称叫“学生”！

经济学有这样一个定理——选择之后必有代价，选择了技术行业，就应该在享受了它相比其他大多数行业的高薪以外，同时承受与之而来的不断学习的压力。不要等到掉队的那天，才反省自己的不作为，给这个行业贴上不该有的标签。其实对于这种人，任何一个需要不断学习的行业，他估计都干不过 30 岁。那些哪怕入门很艰苦，但学成后就不需要再学习，吃老本就可以应付的行业，请放心，它一定是低薪酬的。即使开始一段时间的薪酬很高，也不可能持久。

学习的意义

学习的意义是什么？我想借用一句自己很喜欢的话来说明。“不是说学习本身特了不起，而是学习这个行为意味着你没有完全认同于这个现世和现实，你还有追求，还在奋斗，还有不满，你还在寻找另一种可能性和另一种生活方式。”

我也曾经像许多人一样幻想过，如果能像武侠小说中的主人公，有一位德高望重的前辈把自己的本领毫无保留地倾囊相授，就可以一跃成为技术大牛了！现实生活中确实存在这样的前辈，他们也愿意毫无保留地倾囊相授！可是，现实和武侠小说的最大区别就是，后辈自己没有本事领悟！这是何等的绝望？但确是铁铮铮的现实！

比如，以高德纳老爷子在计算机方面的知识存量和计算机界的地位来说，很难有出其右者，绝对可以称的上是德高望重，但老爷子倾注了毕生所学之精华的《计算机程序设计艺术》，却没有几人能够完全领悟？

我想这便是读书的意义了吧！读书(除去休闲图书)的好处之一便是可以站在比自己牛的人的肩膀上，以勉强理解的视角，去看待自己现阶段不可能遇到，或者经常忽略的问题，给自己大脑的操作系统升级！

我们并不总有机会跟比自己牛逼的人一起共事，所以读书的另一好处就是提供我们和那些牛人一起学习、交流的途径。当然，软件这个行业不止读书这一条路，更好的选择就是看源码！

职业人

程序员们经常戏称自己为码农，作为同事之间的幽默确实挺好，但是我在内心深处一直把自己定位成一名职业人。

大前沿一在他的《专业主义》中，对职业人有过这么一段论述："对上帝发誓，以此为职业的人！"

在这个信仰缺失的年代，加之东方人对上帝本就没有什么敬畏之心，希望我们多少保留些内心的本我！尊重内心中最原始的渴望，不需要借助任何外力的监督，自然而然地培养一种本能！

当然，做到专业的职业人很难，但是却不是不可以后天锻炼形成的。借用猫腻小说《将夜》中宁缺师傅颜瑟的一段话，这种对自己所从事职业的荣誉感可以描述成："修行首重心性，在于敢想敢认，长路漫漫，你自己若都不相信自己能够走到最后，你怎么迈过修行路上那些艰难奇崛的险峰？越优秀的修行者越自信，而最优秀的那些修行者必然自信到极夸张的境界，大概也就是你所说的自恋。"

虽然这是本网络小说，可这句对我的感触最大，记忆也最深刻。回到现实中，我们若不对自己所从事的行业报以极高的荣誉感、认同感，又岂能在自己的行业内有所建树？可以说，我们必须达到极致的自恋，这种职业人的精神反过来会让我们在艰难中不断前行！如果你只是抱着某个行业高薪酬，好找工作，内心对这个行业始终充满不屑的态度，几乎可以肯定，你在这个行业终走不远！

逃离舒适区

罗振宇的节目中曾经有一期对“学习”的定义进行论述，他说读书有三个区域：舒适区、学习区和恐慌区。学习区又称为逃离舒适区！但是，舒适区其实是人类的本能，我们太习惯重复自己，而错过了许多进步的机会！这或许就是牛人之所以牛，而我们现在却一直平庸的原因！

我工作已经四年多了，见过的程序员几乎没有一个愿意逃离自己学习的舒适区，总是在不断地重复着自己之前的技术，积累着所谓的技术经验！但是他们忘了，这个行业的技术迭代速度之高，足以让他们积累的经验变得一文不值！那时的你，在学习能力上又如何跟刚毕业的大学生比？我想，这便是华为开除 34 岁以上那批程序员的真实写照。在爆出华为开除员工消息后，我偶然在微信公众号中看到一位华为员工写的微文《华为之恶》，里面详尽叙述了华为是如何将员工当成螺丝钉用的。虽然我认同这篇文章的部分观点，但决不赞同他的结论！

一个人的职业生涯，一般要远超一个公司的寿命，这其实给我们选择公司时提供了一盏指明灯。对于职业人来说，在选择工作时，薪水是放在

第二位的，而自己的职业发展是第一位的。当然，我必须承认现实有时比较残忍，在这种情况下，我们其实可以先委屈自己一年，但在这一年绝对不要放弃自己的进步。可以抽一定时间自学，不让技术距平均水准太远，相信一年后你绝对有经济基础和经验跳槽！诚然，这对任何人来说都是一个需要勇气才能做出的决定，但这也是你成为职业人，必须要走的道路。华为这样的公司虽然剥夺了你职业生涯的发展，可同时也给予了你超越一般程序员的薪资。你既然没有选择离职，证明你也认可这种协议，等同于你愿意用自己以后的职业生涯去换前几年高于一般同龄人的薪资，那最后被开除了，又有何好抱怨？还是引用经济学的那句话“选择之后，必有代价”，太多的人只愿意接受选择带来的好处，而不愿意承担它所带来的后果！

相较于华为，其实最不可容忍的，还是自己葬送自己的职业生涯。但不幸的是，大多数程序员都是这么干的，我见过太多只愿意重复做自己熟悉技术的程序员。他们对外来的一切新技术都抱有本能的排斥，与其说害怕学习新东西，倒不如说是懒！我们公司的团队里有 3 个刚毕业的大学生，我本来打算自己去培训他们的，好让他们在技术上少走一些弯路，可是尝试过几次后，我彻底放弃了自己的这个想法。因为你永远无法叫醒一个装睡的人！那些不想学习的，他们觉得只要完成工作，单位每月按时发工资就行，虽然他们工作也很辛苦，几乎天天加班，但却根本得不到应该有的回报！

由此，我自己悟出了一个道理：在软件这个行业里，不存在“教会徒弟饿死师傅”的问题。渴望进步的职业人，即使你不告诉他们，他们也迟早也会知道，而那些不愿意去进步的，即使你倾囊相授，掏心掏肺，也

不会有一丝效果。所以，大胆地去分享自己的所学、所思、所悟吧！毕竟我们都是一群有点自傲的孤独者！在这个大多数人都信仰“老板给多少钱，我就干多少活儿”的时代，职业人承受了太多人的不理解，但我相信，时间会给我们最公正的答案！

一次偶然的成功，比一次必然的失败更可怕

我对自己的要求是，在可能的条件下，强迫自己用已经验证的新技术去完成工作。相同的代码，绝不写第二遍，即使第二遍的命名比第一遍规范。

我时刻在内心告诫自己，小心被昨天的经验所束缚。佛门三障之一为“所知障”！我在技术上对它的认识还比较浅薄，总结一下，就是“一次偶然的成功，比一次必然的失败更可怕”。在以前的工作中，我不止一次遇到过这样的情形：曾经用谷歌百度搜索可以解决掉的问题，在第二次遇到时，却无法用之前的方法解决。只有在第二次不能解决时，我才对这个问题产生比较深入的研究和认识。在技术中，我们见过太多的理所应当，而不去探究本源。记得上学时，教我们编程的老师常会说：“这是固定写法，你这么写就对了。”当然，我不是让大家去质疑一切知识本身，而是想告诉大家，在解决或者设计一个问题的解法时，一定要注意避免陷入自己的所知障，敢于打破以前的思维定式！每一次打破，意味着一次的进步！

自己悟出来的方法最好

下面说说自己的学习方法吧！以前的学习方法就是买书，死磕，由于前期的知识存量不足，很难看下去。所以就强迫自己写读书笔记，说好听

是在记笔记，说不好听就是在抄书，因为只有在这种情况下我的精神才能高度集中，渐渐也就爱上这种学习模式！

当有了一定基础之后，笔记的内容就少了，基本上都是花时间去迅速看，了解更多知识！

第三个阶段也是现在的阶段，我重新审视了自己以前的学习方法以及学到的知识，带着解决一个具体问题和批判的角度去重读一些知识点。带着质疑看源码，研究整理了部分的Java基础，看了Spring IOC核心的代码！

在这里，我最想说的不是自己的学习方法。每个人都有自己的学习方法，没有好坏之说。只有你自己开始行动，开始不断地去积累，就会总结出自己独有的学习方法。或许刚开始的方法是最笨的，可是没有之前的笨方法，你就不会在重复一段时间后，找到更好的。与其纠结什么样的学习方法好，还不如现在就开始学。如果你还是要问，什么样的学习方法好？我会回答你，自己悟出来的最好！

学习一门知识，我一般会买至少与这个知识相关的两本书。比如，我最近在看网络相关的内容，就买了图灵的《网络是怎样连接的》和《计算机网络：自顶向下方法》两本书。前一本比较易懂而后者相对比较难点，我会把前一本看到大概一半的位置时，再去看第二本。有了第一本浅显认识的铺垫，阅读第二本时才能比较轻松。看到太懂的章节，我会返回来看第一本，这时我会发现第一本的内容对我来说简直太简单了，会趁机会一口气读完第一本，然后再回到第二本！有时为了学习某技术，我甚至会买5本或者更多，因为每个作者的认知和思路是不同的，对同一

问题的表述也是有一定差异的，这有助于我理解比较难理解的一些概念，同时又能做到内容间的相互印证。

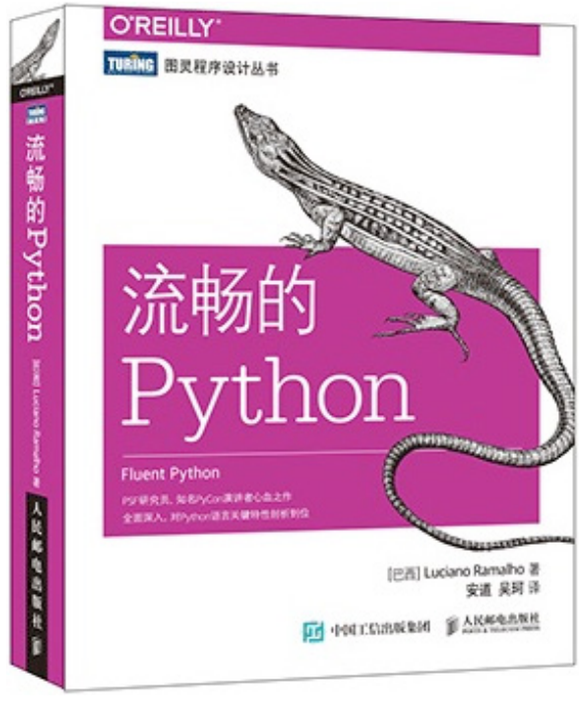
再下一个阶段就是质疑了。孟子说过：“尽信书，则不如无书！”在这一阶段，我们其实已经培养了自己独立的认知，可以去质疑哪怕被冠以权威的东西。这能让我们更容易看清技术的本质，不再被一些技术广告忽悠！在这里，我想引用《Scala 函数式编程》作者的一句类似的话来表述自己的想法：“没有哪个现有的库是权威的，或不需要再去推敲的，哪怕它们都是专家们设计的，还冠以‘标准’的名头。很多库都存在一些随意的设计选择，尽管大多都是无心的。去重新审视那些已存在于设计库中的基本假设，从不同的角度发现别人从未想过的问题。”

既然提到图灵，这里就多说两句，跟图灵社区的结识其实是一件非常偶然的事。第一次得知图灵读者群里的那些牛人时，自己多少是有点不服气的，现在看来当时的自己还是很傲娇的。可是，大家逐渐熟悉起来以后，我不得不承认技术牛人高出我太多，可能尽毕生之力，我也赶不上他们的步伐。他们值得我发自心底的尊重！我是一个对技术很自傲的人，所以这里不存在刻意的逢迎！因为我在进步的时候，他们也在进步，而且进步的速度绝对要比我快！

但借用王安石《游褒禅山记》所说的：“然力足以至焉，于人为可讥，而在己唯有悔；尽吾志也而不能至者，可以无悔矣，其孰能讥之乎？”虽不能至，然吾心向往之！如果因为不能超越而不去努力，只会让更多的人超越自己！况且，职业人首先对自己负责，根本没必要去羡慕别人，追求更好的自己就好！■

选自图灵社区。

书　单



《流畅的Python》

作者：Luciano Ramalho

书号：978-7-115-45415-7

图灵社区推荐：34

本书致力于帮助开发人员挖掘Python及相关程序库的优秀特性，避免重复劳动，同时写出简洁、流畅、易读、易维护的地道Python风格代码。尤其深入探讨了Python语言的高级用法，涵盖数据结构、Python风格的对象、并行与并发，以及元编程等不同的方面。

《数学万花筒》

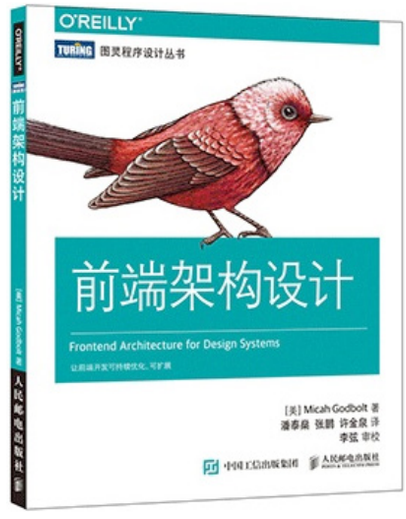
作者：伊恩·斯图尔特

书号：978-7-115-44434-9

图灵社区推荐：15

《数学万花筒》系列是知名数学科普作家、华威大学数学系荣退教授伊恩·斯图尔特（Ian Stewart）五十多年收藏精选，是数学科普畅销经典。

通过本书，读者可以透彻了解数学有趣而迷人的一面，激发想像力；亲自参与到数学中，体验发现带来的兴奋；了解数学的重要发展，不管来自四千年前还是上一周。



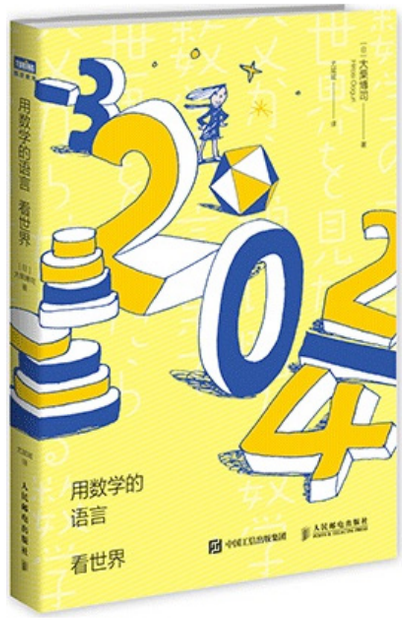
《前端架构设计》

作者：Micah Godbolt

书号：978-7-115-45236-8

图灵社区推荐：13

这是一名成熟的前端架构师对前端开发全面而深刻的理解。作者结合自己在Red Hat公司的项目实战经历，探讨了前端架构的核心内容和原则，包括工作流程、测试流程和文档记录，以及作为前端架构师所要承担的具体开发工作，包括HTML、JavaScript和CSS等。



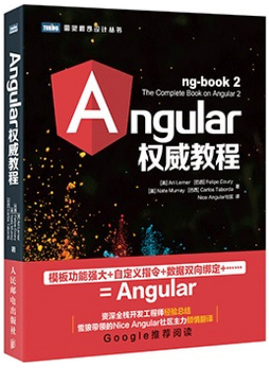
《用数学的语言看世界》

作者：大栗博司

书号：978-7-115-44959-7

图灵社区推荐：6

本书为理论物理学家大栗博司先生写给自己女儿的数学读本，全书以“数学语言”解读自然为线索，用生动的故事和比喻重新讲解了数学的核心原理与体系。



《Angular权威指南》

5

作者：Ari Lerner; Felipe Coury; Nate Murray; Carlos Taborda

书号：978-7-115-45158-3

图灵社区推荐：21

《高性能iOS应用开发》

6

作者：Gaurav Vaish

书号：978-7-115-45120-0

图灵社区推荐：3

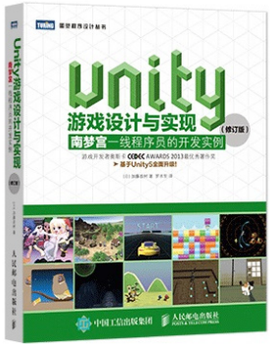
《客户端存储技术》

7

作者：Raymond Camden

书号：978-7-115-45014-2

图灵社区推荐：5



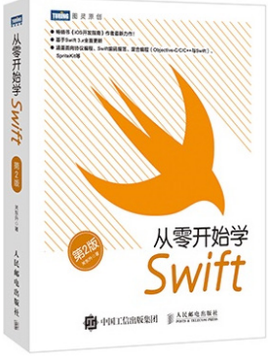
《Unity游戏设计与实现：南梦宫一线程序员的开发实例(修订版)》

8

作者：加藤政树

书号：978-7-115-44899-6

图灵社区推荐：4



《从零开始学Swift（第2版）》

9

作者：关东升

书号：978-7-115-45092-0

图灵社区推荐：3

《图解物联网》

10

作者：NTT DATA集团

书号：978-7-115-45169-9

图灵社区推荐：6

电子书单

1. Angular 权威教程
   —— 堪称 Angular 领域的里程碑式著作，涵盖了关于 Angular 的几乎所有内容。

2. 前端架构设计
   —— 系统总结了前端架构的四个核心，详细展示了新的前端开发准则，将 Web 开发提升到了一个新高度。

3. 高性能 iOS 应用开发
   —— 为有经验的 iOS 开发者提供构建优异应用移动性能所需的开发建议和最佳实践，帮助读者解决常见性能问题。

4. 程序员第二步——从程序员到项目经理
   —— 每一位程序员和项目经理的必读之书！IT 职场至关重要的第二步！

5. 计算进化史：改变数学的命运
   —— 从计算的演变这一独特视角回顾了数学、逻辑学和哲学的历史沿革，展望了计算机科学为数学带来的全新前景，以及由此引发的自然科学与哲学领域的重大变革。

6. Docker 经典实例
   —— 涵盖网络、镜像管理、配置以及包括 Kubernetes 和 Mesos 在内的编排和调度生态系统，对私有云和公有云上部署的应用程序都给出了丰富实用的解决方案和示例。

7. 普林斯顿微积分读本（修订版）
   —— 基于风靡美国的，由普林斯顿大学阿德里安·班纳教授讲解的微积分复习课的一本经典著作！

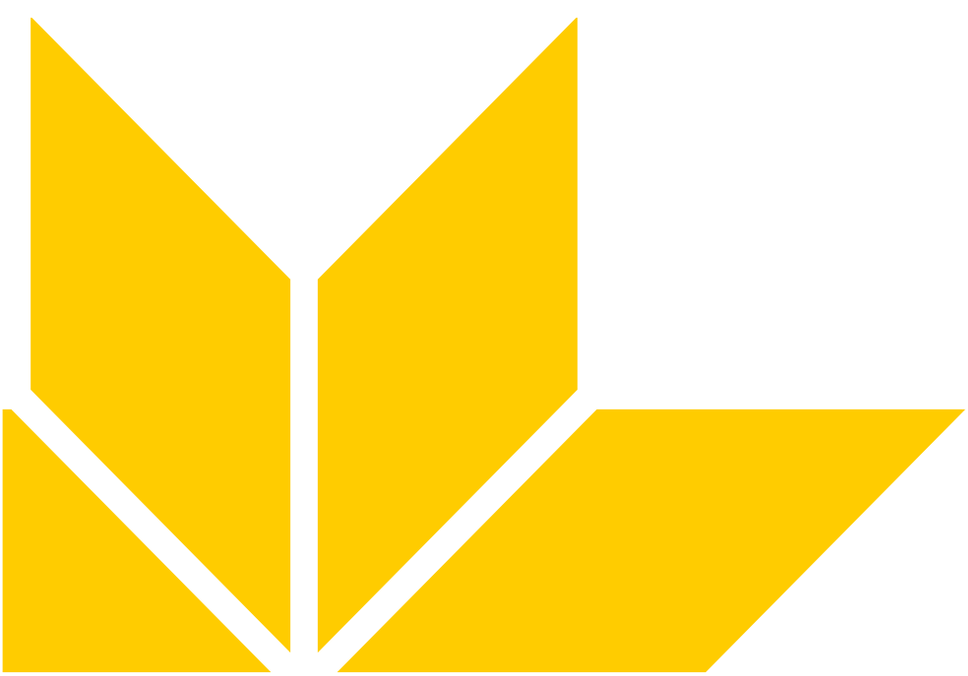
8. 图解性能优化
   —— 从基础知识到最新技术，从系统开发到运维，195 张图解讲透性能！

9. Docker 开发指南
   —— 从基础知识出发，了解如何在多主机系统上运行数十个拥有联网和调度能力的容器系统，重在掌握使用 Docker 来开发、测试以及部署 Web 应用。

10. 挑战编程技能：57 道程序员功力测试题
    —— 基于日常软件开发中经常遇到的实际问题，提炼了 57 道练习题以帮助程序员磨练技艺、提升技能。

图灵社区 出品

出版人：武卫东
编辑：刘敏
设计：大胖



地址：北京市朝阳区北苑路13号院领地OFFICE C座603室
电话：010-51095181
微博：weibo.com/ituring
Email: ebook@turingbook.com