yorker

导读：2007年，受够了C++煎熬的Google首席软件工程师Rob Pike纠集Robert Griesemer和Ken Thompson两位牛人，决定创造一种新语言来取代C++， 这就是Golang。出现在21世纪的GO语言，虽然不能如愿对C++取而代之，但是其近C的执行性能和近解析型语言的开发效率以及近乎于完美的编译速度，已经风靡全球。特别是在云项目中，大部分都使用了Golang来开发，不得不说，Golang早已深入人心。而对于一个没有历史负担的新项目，Golang或许就是个不二的选择。

被称为GO语言之父的Rob Pike说，你是否同意GO语言，取决于你是认可少就是多，还是少就是少(Less is more or lessis less)。Rob Pike以一种非常朴素的方式，概括了GO语言的整个设计哲学--将简单、实用体现得淋漓尽致。

很多人将GO语言称为21世纪的C语言，因为GO不仅拥有C的简洁和性能，而且还很好的提供了21世纪互联网环境下服务端开发的各种实用特性，让开发者在语言级别就可以方便的得到自己想要的东西。

本文大纲：

发展历史

2007年9月，Rob Pike在Google分布式编译平台上进行C++编译，在漫长的等待过程中，他和Robert Griesemer探讨了程序设计语言的一些关键性问题，他们认为，简化编程语言相比于在臃肿的语言上不断增加新特性，会是更大的进步。随后他们在编译结束之前说服了身边的Ken Thompson，觉得有必要为此做一些事情。几天后，他们发起了一个叫Golang的项目，将它作为自由时间的实验项目。

2008年5月 Google发现了GO语言的巨大潜力，得到了Google的全力支持，这些人开始全职投入GO语言的设计和开发。

2009年11月 GO语言第一个版本发布。2012年3月 第一个正式版本Go1.0发布。

2015年8月 go1.5发布，这个版本被认为是历史性的。完全移除C语言部分，使用GO编译GO，少量代码使用汇编实现。另外，他们请来了内存管理方面的权威专家Rick Hudson，对GC进行了重新设计，支持并发GC，解决了一直以来广为诟病的GC时延（STW）问题。并且在此后的版本中，又对GC做了更进一步的优化。到go1.8时，相同业务场景下的GC时延已经可以从go1.1的数秒，控制在1ms以内。GC问题的解决，可以说GO语言在服务端开发方面，几乎抹平了所有的弱点。

在GO语言的版本迭代过程中，语言特性基本上没有太大的变化，基本上维持在GO1.1的基准上，并且官方承诺，新版本对老版本下开发的代码完全兼容。事实上，GO开发团队在新增语言特性上显得非常谨慎，而在稳定性、编译速度、执行效率以及GC性能等方面进行了持续不断的优化。

开发团队

GO语言的开发阵营可以说是空前强大，主要成员中不乏计算机软件界的历史性人物，对计算机软件的发展影响深远。Ken Thompson，来自贝尔实验室，设计了B语言，创立了Unix操作系统（最初使用B语言实现），随后在Unix开发过程中，又和Dennis Ritchie一同设计了C语言，继而使用C语言重构了Unix操作系统。Dennis Ritchie和Ken Thompson被称为Unix和C语言之父，并在1983年共同被授以图灵奖，以表彰他们对计算机软件发展所作的杰出贡献。Rob Pike，同样来自贝尔实验室，Unix小组重要成员，发明了Limbo语言，并且和Ken Thompson共同设计了UTF-8编码，《Unix编程环境》、《编程实践》作者之一。

可以说，GO语言背靠Google这棵大树，又不乏牛人坐镇，是名副其实的“牛二代”。

业务案例

大名鼎鼎的Docker，完全用GO实现，业界最为火爆的容器编排管理系统kubernetes，完全用GO实现，之后的Docker Swarm，完全用GO实现。除此之外，还有各种有名的项目如etcd/consul/flannel等等，均使用GO实现。有人说，GO语言之所以出名，是赶上了云时代，但为什么不能换种说法，也是GO语言促使了云的发展？

除了云项目外，还有像今日头条、UBER这样的公司，他们也使用GO语言对自己的业务进行了彻底的重构。

GO语言之所以厉害，是因为它在服务端的开发中，总能抓住程序员的痛点，以最直接、简单、高效、稳定的方式来解决问题。这里我们并不会深入讨论GO语言的具体语法，只会将语言中关键的、对简化编程具有重要意义的方面介绍给大家，跟随大师们的脚步，体验GO的设计哲学。

GO语言的关键特性主要包括以下几方面：

- 并发与协程
- 基于消息传递的通信方式
- 丰富实用的内置数据类型
- 函数多返回值
- defer机制
- 反射(reflect)
- 高性能HTTP Server
- 工程管理
- 编程规范

并发与协程(goroutine)

通过关键字*go*，语言级别支持协程(微线程)并发，并且实现起来非常简单。

```java
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();
        // ...
    }
}
```

```go
func run(arg st
    // ...
}
func main() {
    go run("tes
    ...
}
```

JAVA多线程并发

GO协程并

在当今这个多核时代，并发编程的意义不言而喻。当然，很多语言都支持多线程、多进程编程，但遗憾的是，实现和控制起来并不是那么令人感觉轻松和愉悦。Golang不同的是，语言级别支持协程(goroutine)并发（协程又称微线程，比线程更轻量、开销更小，性能更高），操作起来非常简单，语言级别提供关键字（go）用于启动协程，并且在同一台机器上可以启动成千上万个协程。

对比JAVA的多线程和GO的协程实现，明显更直接、简单。这就是GO的魅力所在，以简单、高效的方式解决问题，关键字go，或许就是GO语言最重要的标志。

基于消息传递的通信方式

消息通信 - Channel

Channel是GO在语言级别提供给进程内的协程间通信方式，简单易用、线程安全。

通过chan类型实现redis连接池

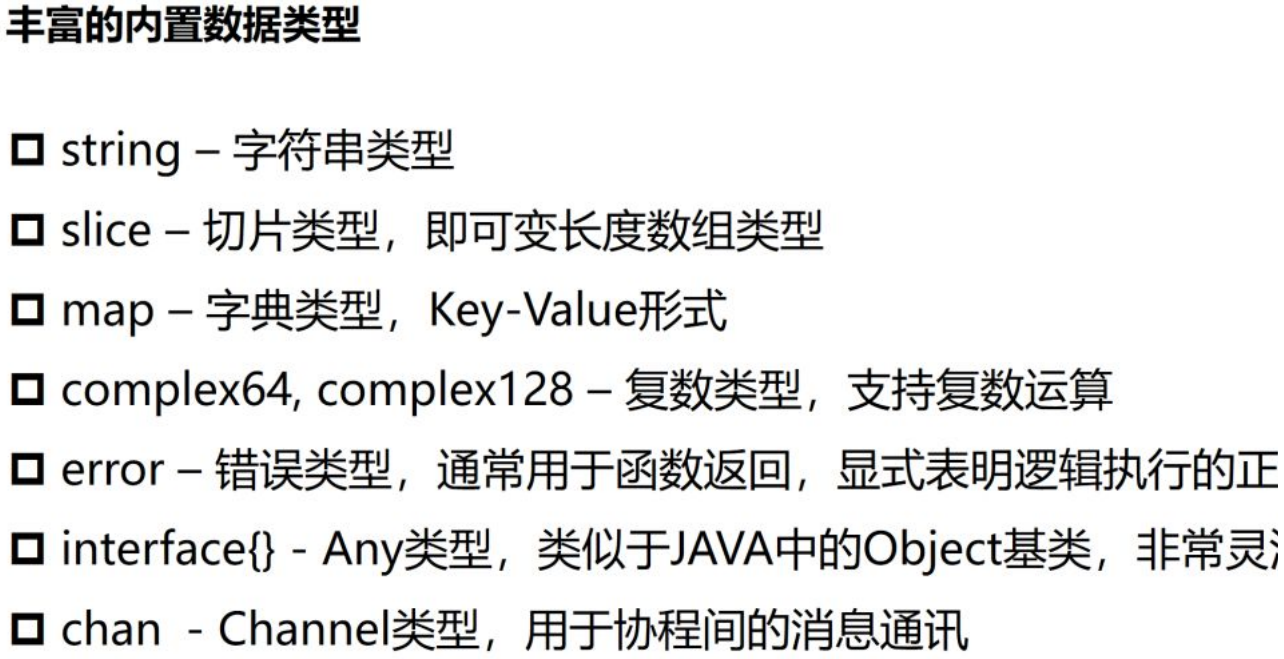
```go
func main() {
    //初始化10个连接
    pool := make(chan redis.Conn, 10)
    for i := 0; i < 10; i++ {
        pool<-redis.Dail(...)
    }

    //10个连接在1000个线程中共享
    for i := 0; i < 1000; i++ {
        go func() {
            //取出连接，如果连接池中没有，则等待
            conn <- pool
            //conn op...
            //使用完毕放回，继续被其它线程使用
            pool <- conn
        }()
    }

    //wait
}
```

通过chan类型实现并发任务的执行等待

```go
func main() {
    //指定启动10个工作任务
    const taskNum int = 10
    //定义一个可存储一定数量的int数据的channel
    chs := make(chan int, taskNum)

    for i := 0; i < taskNum; i++ {
        //启动指定数量的协程数
        go func() {
            //do something

            //此异步逻辑执行完成后，写入标记到
            chs <- 1
        }()
    }
    //这里等待所有的异步逻辑执行完成
    for i := 0; i < taskNum; i++ {
        <-chs
    }

    //继续其它主逻辑
    //TODO
```

在异步的并发编程过程中，只能方便、快速的启动协程还不够。协程之间的消息通信，也是非常重要的一环，否则，各个协程就会成为脱缰的野马而无法控制。在GO语言中，使用基于消息传递的通信方式（而不是大多数语言所使用的基于共享内存的通信方式）进行协程间通信，并且将消息管道(channel)作为基本的数据类型，使用类型关键字(chan)进行定义，并发操作时线程安全。这点在语言的实现上，也具有革命性。可见，GO语言本身并非简单得没有底线，恰恰他们会将最实用、最有利于解决问题的能力，以最简单、直接的形式提供给用户。

Channel并不仅仅只是用于简单的消息通信，还可以引申出很多非常实用，而实现起来又非常方便的功能。比如，实现TCP连接池、限流等等，而这些在其它语言中实现起来并不轻松，但GO语言可以轻易做到。



GO语言作为编译型语言，在数据类型上也支持得非常全面，除了传统的整型、浮点型、字符型、数组、结构等类型外。从实用性上考虑，也对字符串类型、切片类型(可变长数组)、字典类型、复数类型、错误类型、管道类型、甚至任意类型(Interface{})进行了原生支持，并且用起来非常方便。比如字符串、切片类型，操作简便性几乎和python类似。

另外，将错误类型(error)作为基本的数据类型，并且在语言级别不再支持try…catch的用法，这应该算是一个非常大胆的革命性创举，也难怪很多人吐槽GO语言不伦不类。但是跳出传统的观念，GO的开发者认为在编程过程中，要保证程序的健壮性和稳定性，对异常的精确化处理是非常重要的，只有在每一个逻辑处理完成后，明确的告知上层调用，是否有异常，并由上层调用明确、及时的对异常进行处理，这样才可以高程度的保证程序的健壮性和稳定性。虽然这样做会在编程过程中出现大量的对error结果的判断，但是这无疑也增强了开发者对异常处理的警惕度。而实践证明，只要严格按GO推荐的风格编码，想写出不健壮的代码，都很难。当然，前提是你不排斥它，认可它。

函数多返回值

函数多返回值在某些场景下有助于提高代码的描述效率。

```php
function GetServerInfo() {
    $retval = array(
        "ServerIP"=>"xxx",
        "ServerAssetId":"yyy",
        "ServerBusiness":"zzz"
    );
    return $retval;
}

$result = GetServerInfo();
$serverIp = $result["ServerIP"];
$serverAssetId = $result["ServerAssetId"];
$serverBusiness = $result["ServerBusiness"];
```

PHP版本

```go
func GetServerInfo() (svrIp string, svrAsset string, svrBusi
    return "xxx", "yyy", "zzz"
}
serverIp, serverAssetId, serverBusiness := GetServerInfo()
```

GO版本

第二个返回参数为明确的error类型

```go
// Query executes a query that returns rows, typically a SELECT.
func (tx *Tx) Query(query string, args ...interface{}) (*Rows, err
    return tx.QueryContext(context.Background(), query, args...)
}
```

在语言中支持函数多返回值，并不是什么新鲜事，Python就是其中之一。允许函数返回多个值，在某些场景下，可以有效的简化编程。GO语言推荐的编程风格，是函数返回的最后一个参数为error类型（只要逻辑体中可能出现异常），这样，在语言级别支持多返回值，就很有必要了。

Defer延迟处理机制

Defer机制

- defer指定的逻辑在函数体return前或出现panic时执行，非常适合逻辑的善后处理。
- defer机制，很大程度上不仅简化了代码，而且极大的增强了代码的可读性。

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

JAVA版本的资源关闭

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

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

rset := ...
defer rset.Close()

// 正常代码 ...
```

简单明了

GO版本的资源关闭

在GO语言中，提供关键字defer，可以通过该关键字指定需要延迟执行的逻辑体，即在函数体return前或出现panic时执行。这种机制非常适合善后逻辑处理，比如可以尽早避免可能出现的资源泄漏问题。

可以说，defer是继goroutine和channel之后的另一个非常重要、实用的语言特性，对defer的引入，在很大程度上可以简化编程，并且在语言描述上显得更为自然，极大的增强了代码的可读性。

反射

GO语言中的Any类型(interface{})配合简单、强大的类型反射(reflect包)，开发灵活性上接近解析

```
func main() {
    //获取-cmd参数，指定启动的服务类型，如HttpServer等
    //要查看所有提供的服务目录，可通过指定-cmd=GetServices查看
    cmd := flag.String("cmd", "", "cmd is a string indicated what services.")

    //提供给GoServices类中方法的参数，为字符串类型
    //比如HttpServer，可传入http的端口号，指定 -arg=9701
    arg := flag.String("arg", "", "p is a string, parameter for services.")
    flag.Parse()

    if len(*cmd) == 0 {
        flag.Usage()
        return
    }

    //准备好要传入具体逻辑中的参数，通过reflect动态调用
    in := make([]reflect.Value, 0)
    if len(*arg) != 0 {
        in = append(in, reflect.ValueOf(*arg))
    }

    //-cmd参数中指定的所有方法名，均在GoServices类中定义
    //GoServices类中定义的所有public方法，均可做为服务拉起
    service := &services.GoServices{}

    f := reflect.ValueOf(service).MethodByName(*cmd)
    if !f.IsValid() {
        fmt.Printf("-cmd `%s` is invalid.\n", *cmd)
        return
    }

    //通过reflect的形式，根据-cmd参数指定的名称执行特定的逻辑(包括拉起服务、定时任务逻辑、常驻
    f.Call(in)
}
```

通过命令参数动态指定服务调用

如执行 ./gtiop –cmd=HttpServer –arg

通过reflect动态调用GoService类中的H

方法，拉起一个HTTP服务。

如执行 ./gtiop –cmd=DaemonLogToU

reflect动态调用GoService类中的

DaemonLogToUlog方法，拉起一个特定

存服务。

.....

Golang作为强类型的编译型语言，灵活性上自然不如解析型语言。比如像PHP，弱类型，并且可以直接对一个字符串变量的内容进行new操作，而在编译型语言中，这显然不太可能。但是，Golang提供了Any类型(interface{})和强大的类型反射(reflect)能力，二者相结合，开发的灵活性上已经很接近解析型语言。在逻辑的动态调用方面，实现起来仍然非常简单。既然如此，那么像PHP这种解析型语言相比于GO，优势在那里呢？就我个人而言，写了近10年的PHP，实现过开发框架、基础类库以及各种公共组件，虽然执行性能不足，但是开发效率有余；而当遇上Golang，这些优势似乎不那么明显了。

高性能HTTPServer

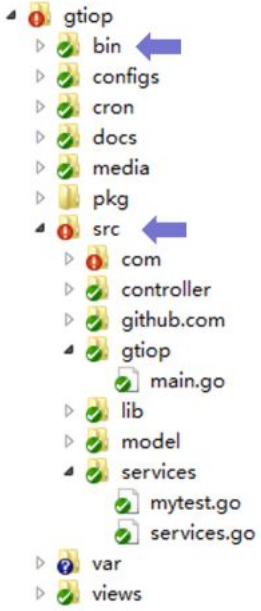
在Golang中，还有一个很重要的能力就是自带高性能HttpServer，通过简单的几行代码调用，就可

一个基于协程的高性能Web服务。更重要的是，维护成本极低，没有任何依赖。

```
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        io.WriteString(w, "hello world.")
    })
    http.ListenAndServe(":8080", nil)
}
```

作为出现在互联网时代的服务端语言，面向用户服务的能力必不可少。GO在语言级别自带HTTP/TCP/UDP高性能服务器，基于协程并发，为业务开发提供最直接有效的能力支持。要在GO语言中实现一个高性能的HTTP Server，只需要几行代码即可完成，非常简单。

工程管理



- 提供GO命令行工具，命令行工具的革命性之处在于彻底消除了工程文件的概念，完全用目录结构和包名来推导工程结构及构建顺序。
- 编译时不需要任何依赖
- 编译速度快
- 编译后生成的二进制文件可以放在同类OS上的任何地方运行，无环境依赖，运维简便。

```
#!/bin/bash
cd $(dirname $0)
curpath=`pwd`
#定位到项目根目录
#并设置根目录到GOPATH
rootpath=`dirname $curpat
export GOPATH=$rootpath
#编译
go build gtiop
```

在GO语言中，有一套标准的工程管理规范，只要按照这个规范进行项目开发，之后的事情（比如包管理、编译等等）都将变得非常的简单。

在GO项目下，存在两个关键目录，一个是src目录，用于存放所有的.go源码文件；一个是bin目录，用于存在编译后的二进制文件。在src目录下，除了main主包所在的目录外，其它所有的目录名称与直接目录下所对应的包名保持对应，否则编译无法通过。这样，GO编译器就可以从main包所在的目录开始，完全使用目录结构和包名来推导工程结构以及构建顺序，避免像C++一样，引入一个额外的Makefile文件。

在GO的编译过程中，我们唯一要做的就是将GO项目路径赋值给一个叫GOPATH的环境变量，让编译器知道将要编译的GO项目所在的位置。然后进入bin目录下，执行go build {主包所在的目录名}，即可秒级完成工程编译。编译后的二进制文件，可以推到同类OS上直接运行，没有任何环境依赖。

编程规范

团队协作，统一的编码规范必不可少，在GO语言中，编码规范强制集成在语言中。

- 命名上，任何需要对外(本包以外)暴露的变量、常量、函数、结构、接口、方法(被外界调用)必须以大写字母开要对外暴露的则以小写字母开头。~~(public, private关键字)~~
- 明确规范代码块花括号的摆放位置，左花括号不允许另起一行。

```
func main() {
    //...
}
```

```
func main()
{
    //...
}
```

- 强制要求一行一句，不允许使用 分号(;)结束语句。
- 不允许导入没有使用的包。
- 不允许定义没有使用的变量。
- 提供gofmt代码格式化工具，所有代码*风格保持一致*。

在Golang中，还能玩出什么花

GO语言的编程规范强制集成在语言中，比如明确规定花括号摆放位置，强制要求一行一句，不允许导入没有使用的包，不允许定义没有使用的变量，提供

gofmt工具强制格式化代码等等。奇怪的是，这些也引起了很多程序员的不满，有人发表GO语言的XX条罪状，里面就不乏对编程规范的指责。要知道，从工程管理的角度，任何一个开发团队都会对特定语言制定特定的编程规范，特别像Google这样的公司，更是如此。GO的设计者们认为，与其将规范写在文档里，还不如强制集成在语言里，这样更直接，更有利用团队协作和工程管理。

编程语言是一个工具，它会告诉我们能做什么，而怎么做会更好，同样值得去探讨。这部分会介绍用GO语言实现的一个开发框架，以及几个公共组件。当然，框架和公共组件，其它语言也完全可以实现，而这里所关注的是成本问题。除此之外，抛开GO语言本身不说，我们也希望可以让大家从介绍的几个组件中，得到一些解决问题的思路，那就是通过某种方式，去解决一个面上的问题，而非一味的写代码，最终却只是解决点上的问题。如果你认可这种方式，相信下面的内容也许会影响你之后的项目开发方式，从根本上提高开发效率。

我们为什么选择GO语言

选择GO语言，主要是基于两方面的考虑

1. 执行性能

缩短API的响应时长，解决批量请求访问超时的问题。在Uwork的业务场景下，一次API批量请求，往往会涉及对另外接口服务的多次调用，而在之前的PHP实现模式下，要做到并行调用是非常困难的，串行处理却不能从根本上提高处理性能。而GO语言不一样，通过协程可以方便的实现API的并行处理，达到处理效率的最大化。

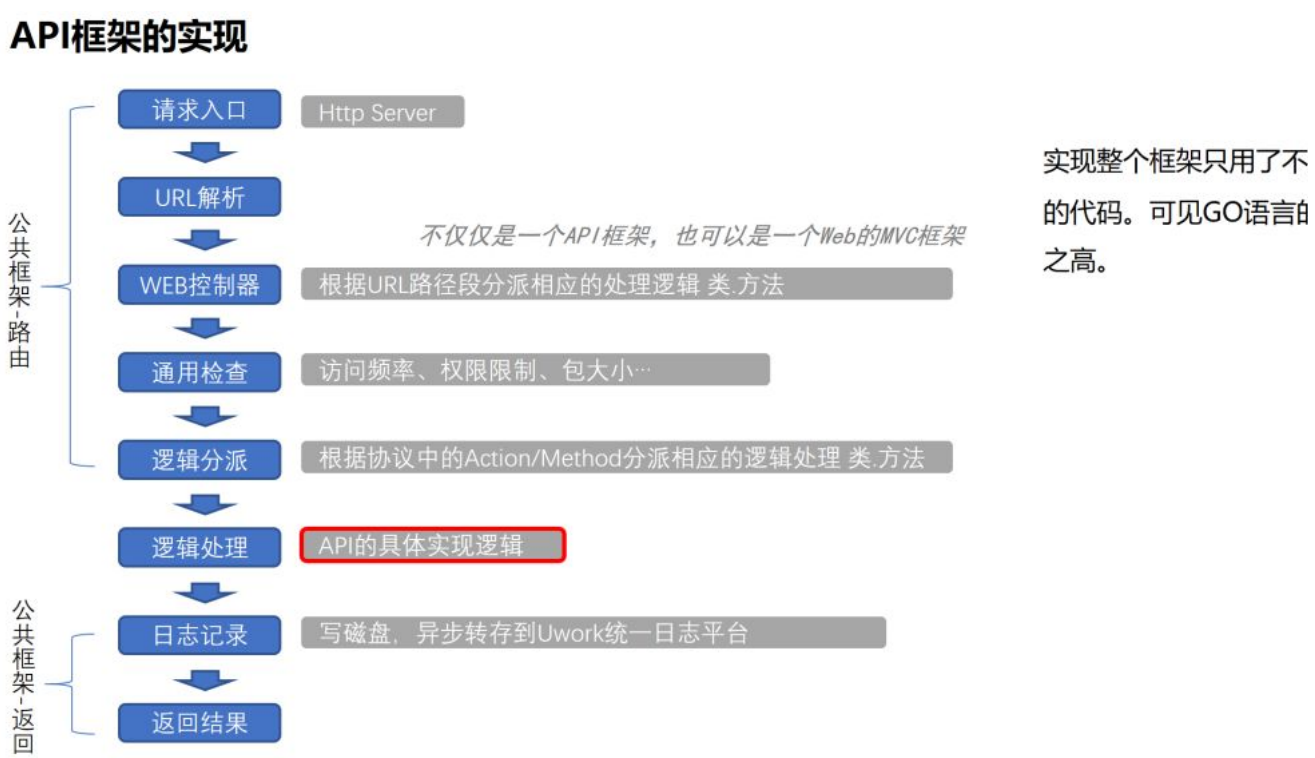
依赖Golang的高性能HTTP Server，提升系统吞吐能力，由PHP的数百级别提升到数千里甚至过万级别。

2. 开发效率

GO语言使用起来简单、代码描述效率高、编码规范统一、上手快。

通过少量的代码，即可实现框架的标准化，并以统一的规范快速构建API业务逻辑。

能快速的构建各种通用组件和公共类库，进一步提升开发效率，实现特定场景下的功能量产。



很多人在学习一门新语言或开启一个新项目时，都会习惯性的是网上找一个认为合适的开源框架来开始自己的项目开发之旅。这样并没有什么不好，但是个人觉得，了解它内部的实现对我们会更有帮助。或许大家已经注意到了，所说的MVC框架，其本质上就是对请求路径进行解析，然后根据请求路径段，路由到相应的控制器（C）上，再由控制器进一步调用数据逻辑(M)，拿到数据后，渲染视图（V），返回用户。在整个过程中，核心点在于逻辑的动态调用。

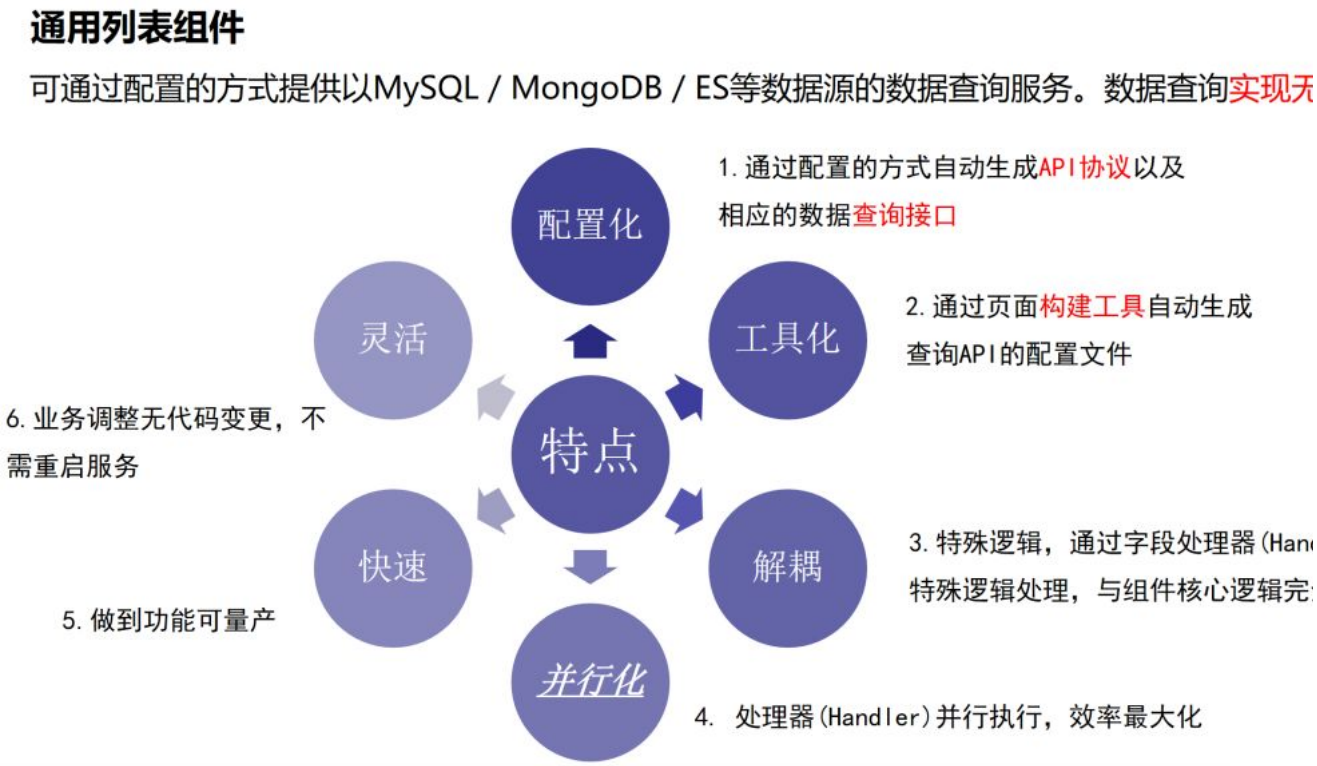
不过，对API框架的实现相对于WEB页面框架的实现，会更简单，因为它并不涉及视图的渲染，只需要将数据结果以协议的方式返回给用户即可。

使用GO语言实现一套完整的MVC开发框架，是非常容易的，集成HTTP Server的同时，整个框架的核心代码不会超过300行，从这里可以实际感受到GO的语言描述效率之高（如果有兴趣，可以参考Uwork开源项目seine）。

也有人说，在GO语言中，就没有框架可言，言外之意是说，引入一个重型的开源框架，必要性并不大，相反还可能把简单的东西复杂化。

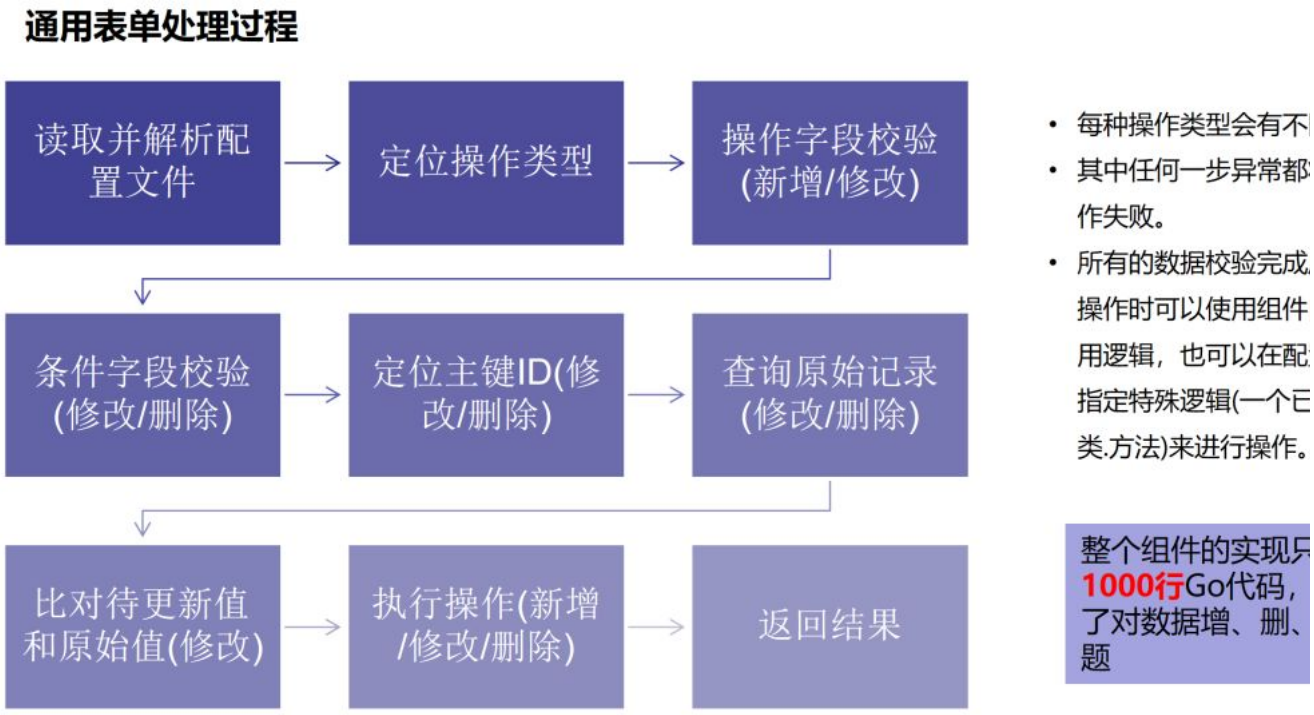
在实际项目开发过程中，只有高效的开发语言还不够，要想进一步将开发效率扩大化，不断的沉淀公共基础库是必不可少的，以便将通用的基础逻辑进一步抽象和复用。

除此之外，通用组件能力是实现功能量产的根本，对开发效率会是质的提升。组件化的开发模式会帮忙我们将问题的解决能力从一个点上提升到一个面上。以下会重点介绍几个通用组件的实现，有了它们的存在，才能真正的解放程序员的生产力。而这些强有力的公共组件在Golang中实现起来并不复杂。同时，结合Golang的并发处理能力，相比于PHP的版本实现，执行效率也会有质的提升。这是组件能力和语言效率的完美结合。

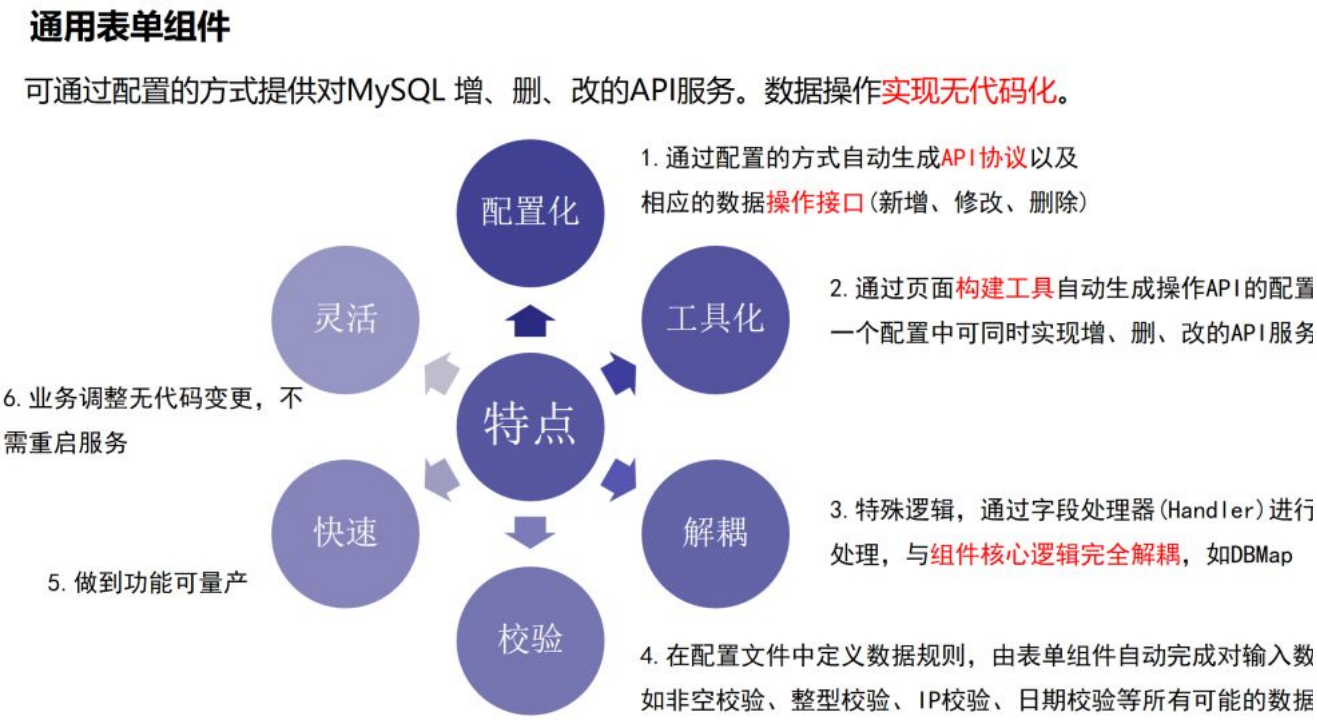


通用列表组件用于所有可能的二维数据源(如MySQL/MongoDB/ES等等)的数据查询场景，从一个面上解决了数据查询问题。在Uwork项目开发中，被大量使用，实现数据查询接口和页面查询列表的量产开发。它以一个JSON配置文件为中心，来实现对通用数据源的查询，并将查询

结果以API或页面的形式自动返回给用户。整个过程中几乎没有代码开发，而唯一要做的只是以一种统一的规范编写配置文件(而不是代码)，真正实现了对数据查询需求的功能量产。

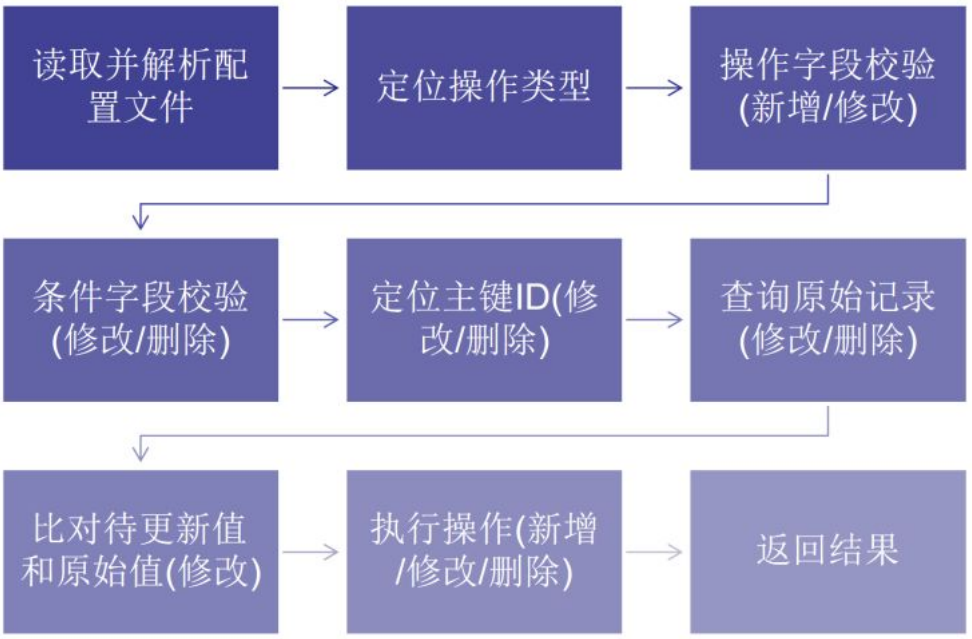


以上是通用列表组件的构建过程，要实现这样一个功能强大的通用组件，是不是会给人一种可望而不可及的感觉？其实并非如此，只要理清了它的整个过程，将构建思路融入Golang中，并不是一件复杂的事情。在我们的项目中，整个组件的实现，只用了不到700行Go代码，就解决了一系列的数据查询问题。另外，通过Golang的并发特性，实现字段处理器的并行执行，进一步的提高了组件的执行效率。可以说，通用列表和Golang的融合，是性能和效率的完美结合。



通用表单组件主要用于对数据库的增、删、改场景。该组件在Uwork的项目开发中，也有广泛的应用，与通用列表类似，以一个JSON配置文件为中心，来完成对数据表数据的增、删、改操作。特别是近期完成的部件级SDB管理平台，通过通用表单实现了对整个系统的数据维护，通过高度抽象化，做到了业务的无代码化生产。

通用表单处理过程



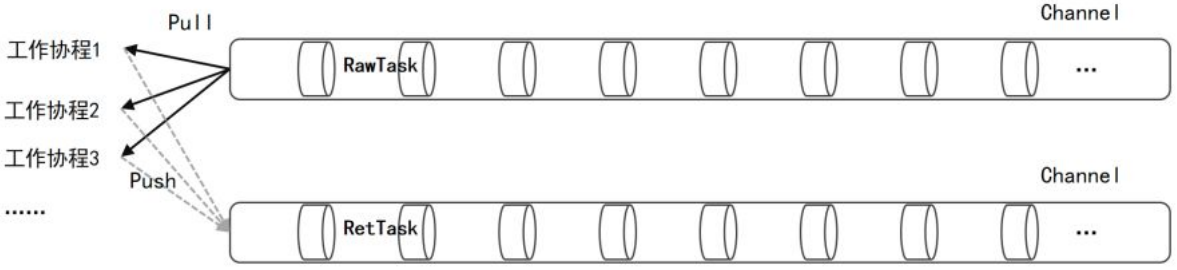
- 每种操作类型会有不
- 其中任何一步异常都
  作失败。
- 所有的数据校验完成
  操作时可以使用组件
  用逻辑，也可以在配
  指定特殊逻辑(一个已
  类.方法)来进行操作。

整个组件的实现只
1000行Go代码，
了对数据增、删、
题

以上是通用表单的完整构建过程，而对于这个一个组件的实现，我们用了不到1000行的GO代码，就解决了对数据表数据维护整个面上的问题。

协程池

Golang原生支持协程并发，可以非常容易的同时启动成千上万个工作协程。但如何方便的控制协程的量，即可以实现效率最大化，又不对外部服务造成冲击，就需要实现一套协程池机制。



- 在Golang中，chan是语言级支持的一种数据类型，实现了协程间基于消息传递的通信方式。
- chan往往是全局的，在所有协程之间共享，但对chan类型的操作是线程安全的。
- 所有的工作协程以抢占式的方式从chan中获取数据并完成处理。
- 当工作协程检测到chan数据长度为0时，退出并返回主逻辑。

协程池的实现基于原
和channel机制，只
Go代码，即完成了对
的通用实现。

GO语言本身支持协程并发，协程非常轻量，可以快速启动成千上万个协程工作单元。如果对协程任务的数量控制不当，最后的结果很可能适得其反，从而对外部或本身的服务造成不必要的压力。协程池可以在一定程度上控制执行单元的数量，保证执行的安全性。而在Golang中要实现这样一个协程池，是非常简单的，只需要对channel和goroutine稍加封装，就可以完成，整个构建过程不到80行代码。

数据校验

在UworkAPI中，可以通过配置化的方式来完成数据校验，达到数据校验去代码化的目的。配置的API协议定义机制(DDP)类似，可以做到对任意的数据结构进行数据校验，这种方式比较适合复杂构的递归校验

- 字段非空校验。
- 基本类型校验，如string, int, float
- 特殊类型校验，如ip, date, datetime, time, email
- 范围校验，如传入的值必须在指定的数据列表范围内
- 逻辑比较校验，如 >, < >=, <=
- 数据结构的层次递归校验

```
ddp := lib.NewDDProto()
if err := ddp.Parse("Module"); err != ni
    //parse template error handle
}
err := ddp.CheckData(jsonData, "Module",
if err != nil {
    //check fail handle
}
```

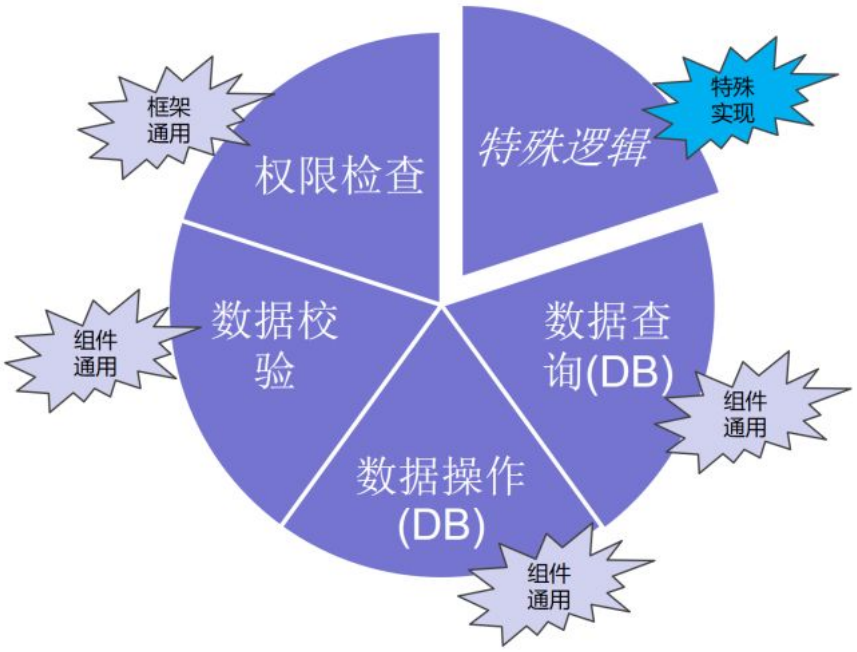
整个组件的实现只用了700行Go代码，就解决了对复杂请求数据的通用校验。

在API开发过程中，数据校验永远是必不可或缺的一个环节。如果只是简单的数据校验，几行代码也许就完成了，可是当遇上复杂的数据校验时，很可能几百行的代码量也未必能完成，特别是遇到递归类型的数据校验，那简直就是一个噩梦。

数据校验组件，可以通过一种数据模板的配置方式，使用特定的逻辑来完成通用校验，开发者只需要配置好相应的数据模板，进行简单的调用，即可完成整个校验过程。而对于这样一个通用性的数据校验组件，在GO语言中只用了不到700行的代码量就完成了整个构建。

小结

业务开发效率



在大多数的后台API系统中DB的增、删、改、查以及的合法性校验，开发成本基省掉了一大半。
而对于数据管理系统来说，发成本是极低的。

在实际项目开发过程中，对开发效率提升最大的，无疑是符合系统业务场景的公共组件能力，这点也正好应证了Rob Pike那句话（Less is lessor Less is more），真正的高效率开发，是配置化的，并不需要写太多的代码，甚至根本

就不需要写代码，即可完成逻辑实现，而这种方式对于后期的维护成本也是最优的，因为做到了高度的统一。

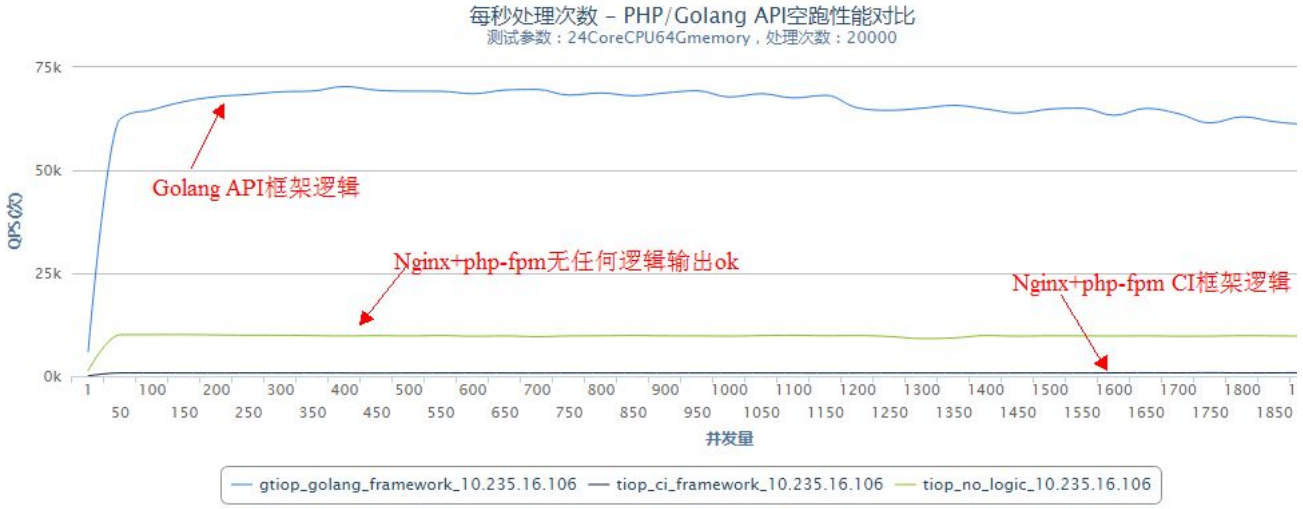
GO的语言描述效率毋庸置疑，对上述所有公共组件的实现，均未超过1000行代码，就解决了某个面上的问题。

(以上的部分代码已经在Uwork开源项目seine中提供)

压力测试环境说明：

- 服务运行机器：单台空闲B6，24核CPU、64G内存。
- PHP API环境：Nginx+PHP-FPM，CI框架。其中Nginx启动10个子进程，每个子进程最大接收1024个连接，php-fpm使用static模式，启动2000个常驻子进程。
- Golang API环境：使用go1.8.6编译，直接拉起Golang API Server进程(HttpServer)，不考虑调优。
- 客户发起请求测试程序：使用Golang编写，协程并发，运行在独立的另外一台空闲B6上，24核CPU，64G内存，依次在1-2000个不同级别(并发数步长为50)的并发上分别请求20000次。

压力测试结果对比



在Golang API框架中，当并发数>50时，处理QPS在6.5w/s附近波动。表现稳定，压力测试过程无报错。

Nginx+php-fpm，只在index.php中输出exit('ok')，当并发数>50时，处理QPS在1w/s附近波动。表现稳定，压力测试过程无报错。

Nginx+php-fpm+CI框架中，逻辑执行到具体业务逻辑点，输出exit('ok')，当并发数>50时，处理QPS在750/s附近波动。并且表现不稳定，压力测试过程中随着并发数的增大，错误量随之增加。

通过压力测试可以发现，Golang和PHP在执行性能上，并没有什么可比性；而使用Golang实现的HTTP API框架，空载时单机性能QPS达到6.5w/s，还是非常令人满意的。

以下是在实际开发过程中遇到的一些问题，仅供参考：

1. 异常处理统一使用error，不要使用panic/recover来模拟throw...catch，最初我是这么做的，后来发现这完全是自以为是的做法。
2. 原生的error过于简单，而在实际的API开发过程中，不同的异常情况需要附带不同的返回码，基于此，有必要对error再进行一层封装。
3. 任何协程逻辑执行体，逻辑最开始处必须要有defer recover()异常恢复处理，否则goroutine内出现的panic，将导致整个进程宕掉，需要避免部分逻辑BUG造成全局影响。
4. 在Golang中，变量(chan类型除外)的操作是非线程安全的，也包括像int这样的基本类型，因此并发操作全局变量时一定要考虑加锁，特别是对map的并发操作。
5. 所有对map键值的获取，都应该判断存在性，最好是对同类操作进行统一封装，避免出现不必要的运行时异常。
6. 定义slice数据类型时，尽量预设长度，避免内部出现不必要的数据重组。