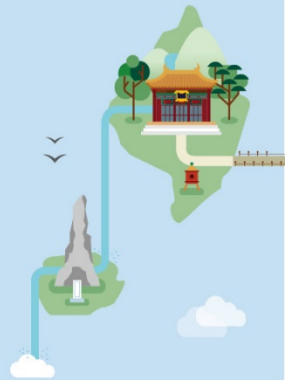
# Go语言在证券期货行情系统中的实践

金大师 张泽武



# 目录

CONTENTS

项目故事

行情系统

接入服务

# 项目故事

- 项目启动
- 团队组建
- 项目计划

# 启动

## 开发一套行情系统

**最短的时间交付？**
**满足大量并发请求？**
**低延时？**
**指标、计算服务？**

- 接入二级平台或交易所的数据
- 提供高速实时行情数据服务
- 提供分时、K线、指标等数据服务
- 接入服务单节点并发10000
- 十一假前交付
- 3个月的开发时间
- 组建团队

# 团队

## 组建一个团队

**团队的主开发语言？**
**证券、期货经验？**
**服务端开发经验？**
**炒股吗？**

- 有证券、期货从业经验
- C++
- 有服务器开发经验
- 自己炒过股的
- 有golang开发经验或喜欢golang的

# 团队

## 组建一个团队

**团队的主开发语言？**
**证券、期货经验？**
**服务端开发经验？**
**炒股吗？**

15.06.09

### C++工程师

2年证券C++开发经验
1年证券C++服务端开发经验

15.06.29

### C++工程师

3年C++服务端开发经验
无Golang，无证券、期货行情经验

### C++工程师

7年C++开发经验
无服务端开发经验
无Golang，无证券、期货行情经验

15.08.10

### C++工程师

3年游戏C++服务端开发经验
无Golang开发经验

# 计划

## 开发一套行情系统

马上启动

尽快招人

对接数据服务商

开发框架

2015.05 **启动**
讨论业务框架，确定需求范围，工期

2015.06 **组建团队**
招人，业务接洽，技术方案、框架准备

2015.07 **研发**
基础服务开发，数据接入

2015.08 **集成、联调**
服务集成，联调测试

2015.09 **预发布**
测试、发布

…… ……

# 行情系统

- 行情系统
- 基本要求
- 服务设计
- 框架设计

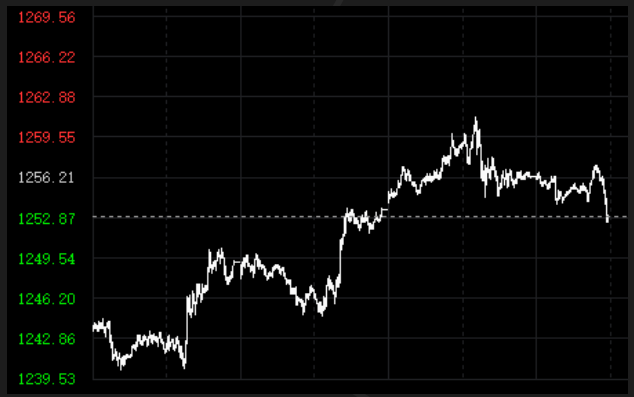
# 行情

## 行情是什么？

合约品种的即时报价
合约品种的历史报价统计
合约品种的历史报价的加工处理

1. 需求：上菜场买土豆
2. 询价：老板，土豆多少钱一斤？
3. 报价：土豆3块一斤？
4. 行情：老板，报价变了通知我一下
5. 并发：好多人同时问老板
6. 服务：问完土豆，还有青菜、黄瓜...

# 行情服务的基本要求

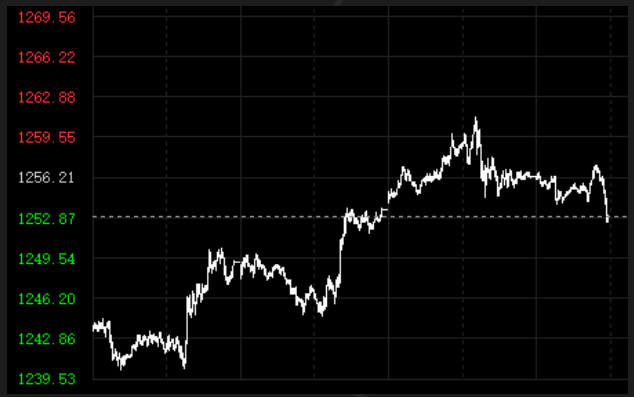



**快** 行情要足够快
响应快，延时低

**准** 数据不能有偏差
处理要准确

**稳** 服务要稳定
可用性99.99%

# 系统设计的特性要求




**并发** 当随着用户规模变大时，系统应该要具备弹性伸缩能力

**容错** 一旦出现故障时，系统能否把故障的影响范围控制在局部，不影响整体服务

**故障响应** 一旦出现故障，且无法控制在局部时，能不能快速恢复？

# 服务设计

框架内部服务化，使得各模块专注于自身的服务定位，易于实现

**接入服务**

面向客户端
实现高并发、高在线的需求

**计算服务**

加工行情数据
按特性组织管理数据

**采集服务**

从行情源采集数据
异构数据转换为同构数据

# 框架设计

开发框架化，抽象出基本服务进行go库化，既然快速开发，同时还能统一管理服务

1. main.go 服务启动器
2. Frame 行情应用框架
3. Status 统计库
4. Command 命令框架
5. Dispatcher 调度框架
6. Config 配置库
7. utils 工具库
8. 基础业务库


1 解耦

2 对象化/组件化

3 服务化

# 基础业务库

开发人员可以专注在业务需求的开发实现上
其他语言转Golang的技术难度也能大幅降低

1. 协议库
2. 第三方数据源go化封装及协议转换库
3. 交易日处理库
4. 多路行情源竞争库
5. 分时、K线、逐笔、分价算法库
6. 指标算法库
7. 行情数据压缩算法库
8. 数据缓存策略库
9. 其他，根据需求增加……

# 接入服务

- 服务设计
- 服务去状态
- 故障恢复
- 负载均衡
- 测试数据

# 接入服务

## 稳定及时的优质服务

如何长期保持接入业务稳定？
如何保证及时的服务响应？
当服务异常时使客户端无感？
如何支持新增服务弹性上线？

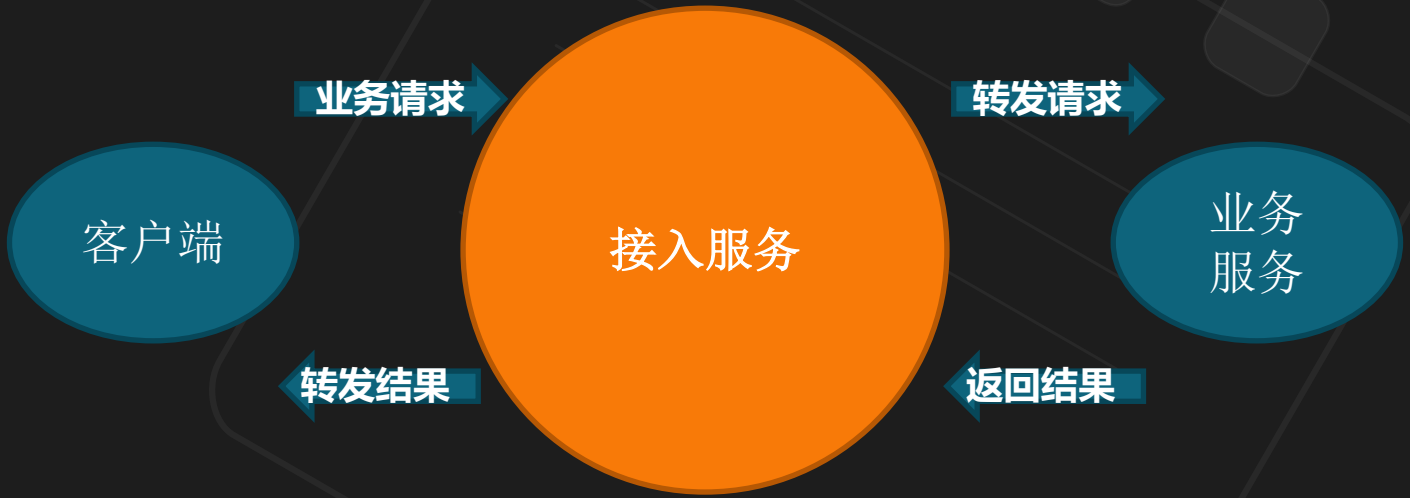
- 解耦业务，与具体业务无关，固化接入服务，保证接入服务在完成交付后，持续稳定
- 提供服务注册功能，由后端业务服务程序注册到接入服务，任何业务都可以注册到接入服务
- 可同时注册多个同一业务的服务程序，并提供多播策略和多策略负载均衡服务
- 提供业务路由服务，按请求协议中的业务路由到服务提供者
- 提供业务状态寄存服务，自动收录注册成功的业务服务的上下文状态，当服务异常时转发至正常服务节点用于恢复服务，使客户无感
- 支持路由策略、多播策略、负载均衡策略的扩展

# 接入服务设计

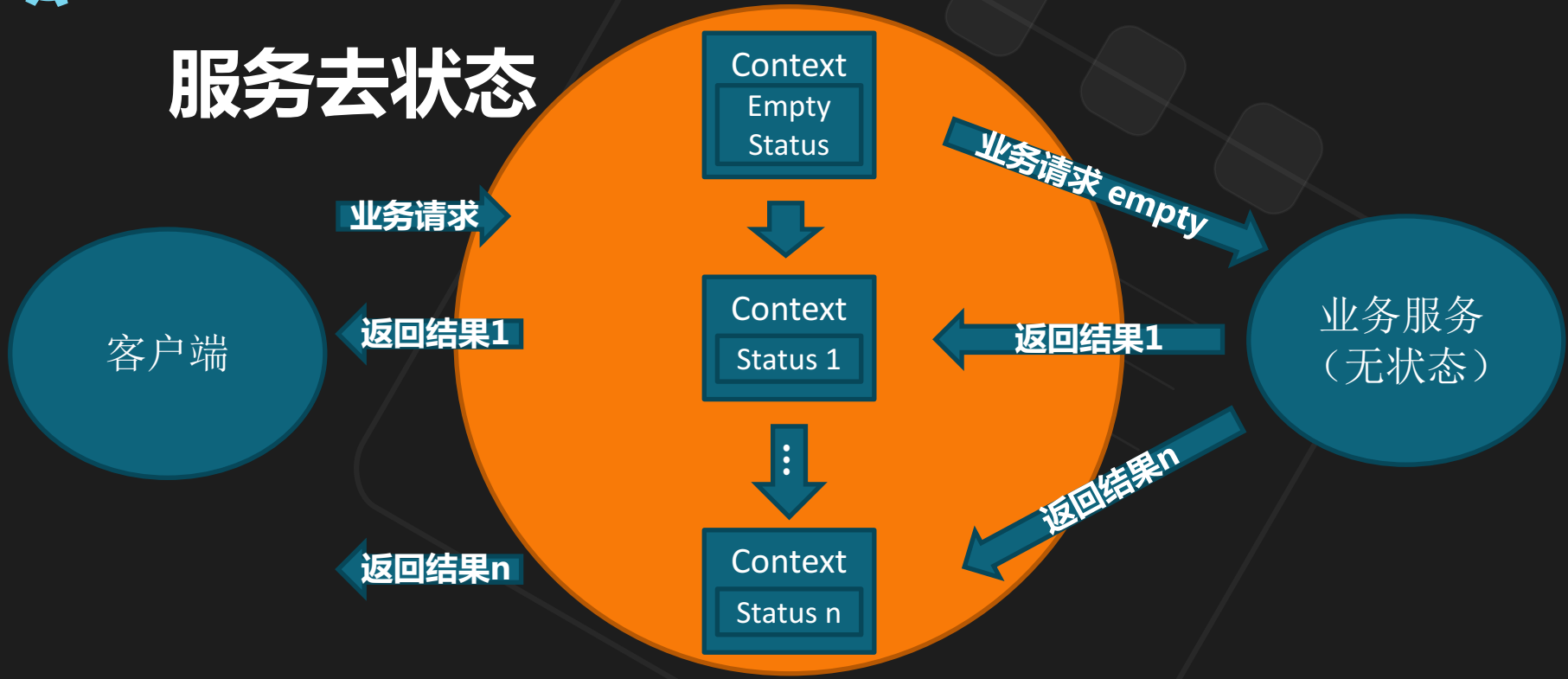
转发服务
服务去状态
故障恢复
负载均衡
策略控制

1. Network 收、发网络模块
2. Process 服务调度模块
3. Route 路由模块
4. Service 服务管理模块
5. Context 状态寄存模块
6. Command 命令处理模块

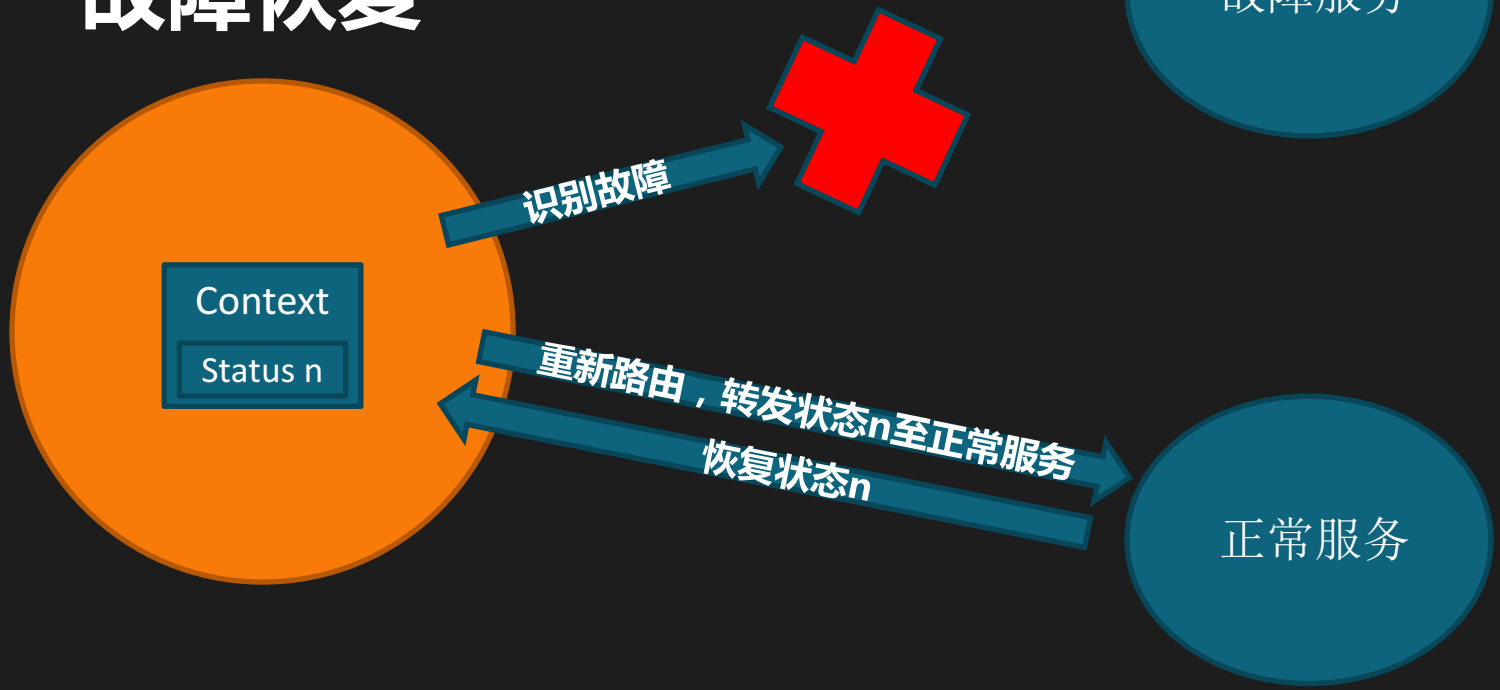
# 转发服务



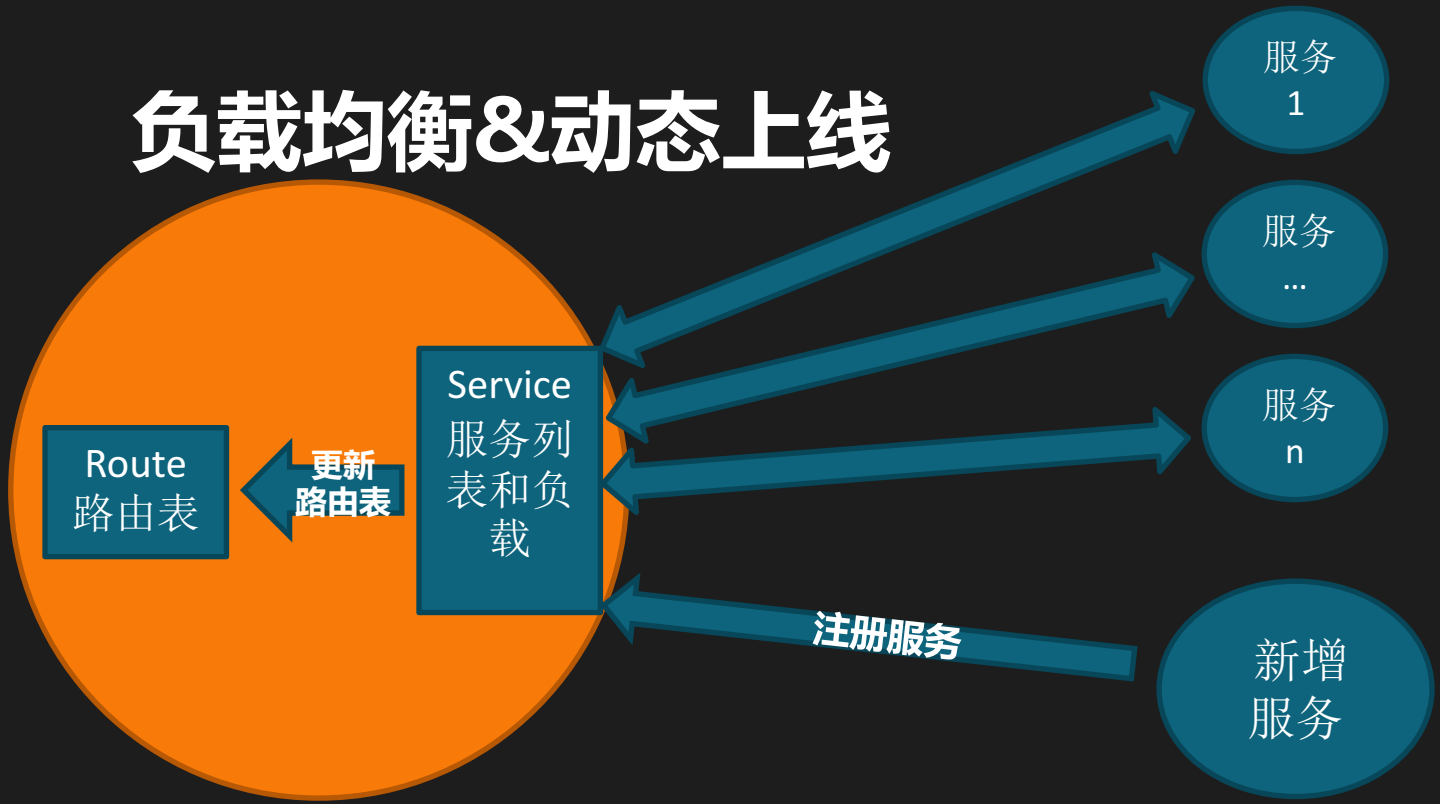
# 服务去状态


客户端
业务请求
返回结果1
返回结果n
Context
Empty
Status
Context
Status 1
Context
Status n
业务请求 empty
返回结果1
返回结果n
业务服务
（无状态）

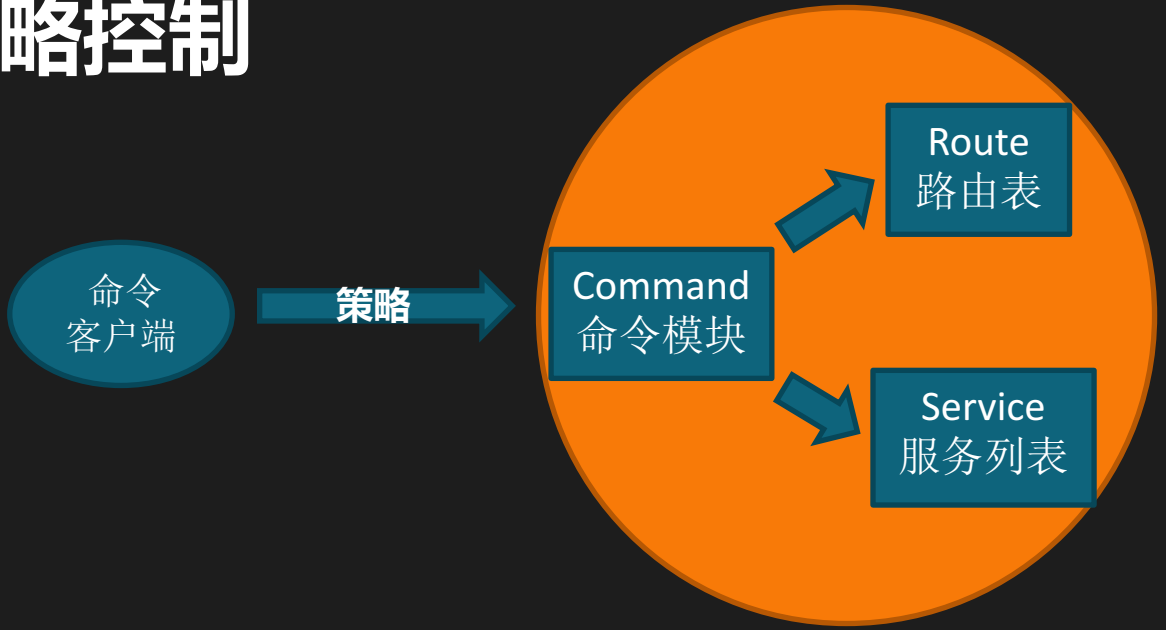
# 故障恢复



# 负载均衡&动态上线



# 策略控制



# 测试场景

| 场景 | 说明 |
| --- | --- |
| 连接数 | 能承载的并发client连接数量 |
| 接收模块能力 | 并发接收client数据包数量 |
| 接收+发送模块能力（少客户端） | 并发接收client数据包，通过路由，再发送到对应server端；（客户端50以内） |
| 接收+发送模块能力（多客户端） | 与上述相比，客户端数量为5000,10000,20000等 |
| 接收+发送+回复 | 并发接收client数据包，通过路由，再发送到对应server端，server处理并返回，再转发至client |
| 数据延时 | 延时数据统计 |
| 队列 | 为了测试内部队列数量对性能的影响，做了相关的测试 |

本测试使用的数据包大小为200 Byte，在1000mbps网络环境下

# 示例

Client

```
request := &jcproto.BaseHead{}
request.MsgID = Enum_Quote_Dyna
……
protobytes, err := proto.Marshal(request)
……
net.Send(protobytes)
```

Server

```
dynaserver, err := acssdk.NewServer()
dynaserver.Register("192.168.8.100:8888",
                    "DynaServer",
                    Enum_Quote_Dyna)
……
dynaserver.Work(protoChannel)
```

```
readdata, err := <-protoChannel
……
protorequest := &jcproto.BaseHead{}
proto.Unmarshal(readdata, protorequest)
```

# 接收能力


| 场景 | 1接收，pkg/s | 1接收，Mb/s | 队列 | CPU | 内存，MB | 延时，s | Go ver. | 说明 |
|---|---|---|---|---|---|---|---|---|
| 接收能力测试 | 500,000+ | 900+ | 4&8&16 | 640/800 | 80 | 0 | 1.4 | 达到网卡上限 |
| | 500,000+ | 900+ | 4&8&16 | 400/800 | 80 | 0 | 1.6 | 1.6版本下，CPU耗费降低了200 |

# 转发测试


| 场景 | 接收pkg/s | 1接收Mb/s | 2发送Mb/s | 队列 | CPU | 内存 MB | 延时 s | Go ver. | 说明 |
|---|---|---|---|---|---|---|---|---|---|
| 接收&发送能力测试 | 250,000+ | 400+ | 400+ | 4&8&16 | 690/800 | 50 | 0 | 1.6 | CPU 达上限* |
| | 500,000+ | 900+ | 900+ | 24 | 1300/2400 | 100 | 0 | 1.6 | 网络达上限* |

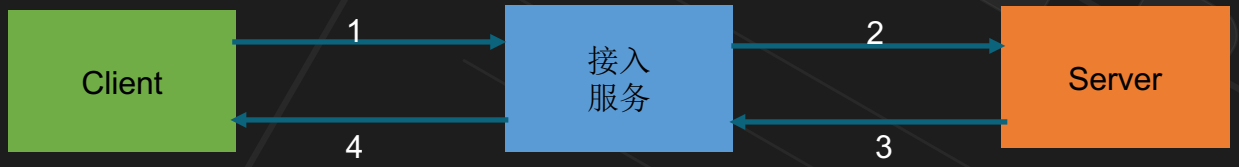
*4核8线程，由于其他服务也会运行于测试服务器上，占据一个线程左右。所以700左右的CPU已经到达到达瓶颈，此时网络流量为400+，未达到网卡上限。

* 12核24线程机器，网络先到达瓶颈，无内存积压，无延时，数据几乎实时到达服务端。

# 多客户端测试

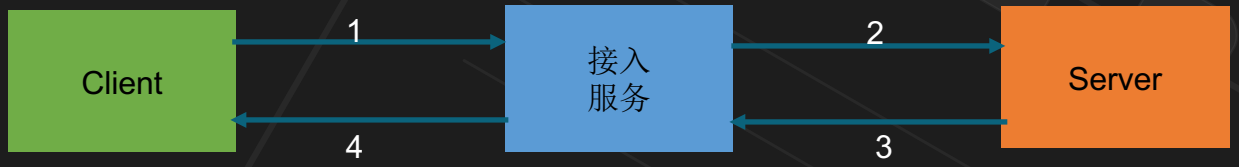

| 场景 | Client | Pkg/s | 1接收Mb/s | 2发送Mb/s | 队列 | CPU /800 | 内存 MB | 延时 s | Go ver. | 说明 |
|---|---|---|---|---|---|---|---|---|---|---|
| 接收发送能力测试 | 5000 | 10 | 100 | 100 | 8 | 150 | 50 | - | 1.6 | |
| | 10000 | 10 | 200 | 200 | 8 | 300 | 100 | - | | |
| | 10000 | 20 | 380 | 380 | 8 | 580 | 2.4G | - | | |
| | 20000 | 10 | 360 | 360 | 8 | 580 | 4G | - | | |
| | 10000 | 25 | 430 | 430 | 8 | 630 | 6.3G | - | | CPU 达上限 |
| | 25000 | 10 | 430 | 430 | 8 | 630 | 7G | - | | CPU 达上限 |

# 请求应答测试




| 场景 | 1 接收 client Mb/s | 2发送 server Mb/s | 3接收 server Mb/s | 4发送 client Mb/s | 队列 | CPU | 内存 MB | 延时 s | Go Ver. | 说明 |
|---|---|---|---|---|---|---|---|---|---|---|
| 接收回复 | 450+ | 450+ | 450+ | 450+ | 24 | 1300/2400 | 100 | 0* | 1.6 | *达到网卡上限 |

# 数据延时




```
2016/06/13 18:08:48 Send Req [ Msg_Request ], current  1000011
2016/06/13 18:08:48 2016-06-13 18:08:48.356998672 +0800 CST
2016/06/13 18:08:48 recv Req [ Msg_Response ], current  1000011
2016/06/13 18:08:48 2016-06-13 18:08:48.566210395 +0800 CST
2016/06/13 18:08:48 Send Req [ Msg_Request ], current  1100012
2016/06/13 18:08:48 2016-06-13 18:08:48.676477468 +0800 CST
2016/06/13 18:08:48 recv Req [ Msg_Response ], current  1100012
2016/06/13 18:08:48 2016-06-13 18:08:48.996515442 +0800 CST
2016/06/13 18:08:48 Send Req [ Msg_Request ], current  1200013
2016/06/13 18:08:49 2016-06-13 18:08:49.149927737 +0800 CST
2016/06/13 18:08:49 recv Req [ Msg_Response ], current  1200013
2016/06/13 18:08:49 2016-06-13 18:08:49.378615589 +0800 CST
2016/06/13 18:08:49 Send Req [ Msg_Request ], current  1300014
2016/06/13 18:08:49 2016-06-13 18:08:49.656348489 +0800 CST
2016/06/13 18:08:49 recv Req [ Msg_Response ], current  1300014
2016/06/13 18:08:49 2016-06-13 18:08:49.860967943 +0800 CST
2016/06/13 18:08:49 Send Req [ Msg_Request ], current  1400015
2016/06/13 18:08:50 2016-06-13 18:08:50.013954643 +0800 CST
2016/06/13 18:08:50 recv Req [ Msg_Response ], current  1400015
2016/06/13 18:08:50 2016-06-13 18:08:50.393270537 +0800 CST
2016/06/13 18:08:50 Send Req [ Msg_Request ], current  1500016
```

基于上例中模型(CPU 1300，网络流量到上限)，每10W数据包统计一下时间点，记录从Client发出到收到回复的时间。在上百万包发送后均无延时出现（1s内）

# 队列数量

| 场景 | Client | 1接收 client Mb/s | 2发送 server Mb/s | 3接收 server Mb/s | 4发送 client Mb/s | 队列 | CPU /800 | 内存 MB | 延时 s | Go Ver. | 说明 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 接收回复 | 2500个 200B/pkg 10pkg/s | 50+ | 50+ | 50+ | 50+ | 1 | 180-190 | 120 | 0 | 1.6 | CPU使用率随队列变大下降，内存占用上升 |
| | | 50+ | 50+ | 50+ | 50+ | 4 | 170-180 | 140 | 0 | | |
| | | 50+ | 50+ | 50+ | 50+ | 8 | 160-170 | 160 | 0 | | |
| | | 50+ | 50+ | 50+ | 50+ | 48 | 140-160 | 360 | 0 | | |
| | | 50+ | 50+ | 50+ | 50+ | 100 | 140-160 | 600 | 0 | | |
| | | 50+ | 50+ | 50+ | 50+ | 800 | 130-140 | 3.8G | 0 | | |
| | 5000个 200B/pkg 10pkg/s | 100+ | 100+ | 100+ | 100+ | 1 | 350 | 220 | 0 | 1.6 | CPU使用率随队列变大下降，内存占用上升 |
| | | 100+ | 100+ | 100+ | 100+ | 4 | 350 | 260 | 0 | | |
| | | 100+ | 100+ | 100+ | 100+ | 8 | 350 | 300 | 0 | | |
| | | 100+ | 100+ | 100+ | 100+ | 48 | 300 | 500 | 0 | | |
| | | 100+ | 100+ | 100+ | 100+ | 100 | 300 | 600 | 0 | | |
| | | 100+ | 100+ | 100+ | 100+ | 800 | 250 | 3.9G | 0 | | |
| | 10000个 200B/pkg 10pkg/s， | 100+ | 100+ | 100+ | 100+ | 1 | 450-520 | 680 | 0 | 1.6 | 队列成为瓶颈 |
| | | 100+ | 100+ | 100+ | 100+ | 4 | 550-600 | 1.4G | 0 | | |
| | | 150+ | 150+ | 150+ | 150+ | 8 | 600-650 | 900 | 0 | | CPU使用率随队列变大下降，内存占用上升 |
| | | 150+ | 150+ | 150+ | 150+ | 48 | 550-600 | 800 | 0 | | |
| | | 150+ | 150+ | 150+ | 150+ | 100 | 450-550 | 990 | 0 | | |
| | | 150+ | 150+ | 150+ | 150+ | 800 | 450-550 | 4.1G | 0 | | |


微信： zewuxx

# 感谢聆听