**GoOne服务器架构说明文档V1.0**

|  |  |  |  |
| --- | --- | --- | --- |
| **版本号** | **编辑时间** | **作者** | **内容** |
| V1.0 | 2020年6月 |  | 基础内容 |
|  |  |  |  |
|  |  |  |  |

[1. 环境 4](#_Toc2018332661)

[1.1 软件需求 4](#_Toc465345415)

[1.2 硬件 4](#_Toc2072431178)

[1.3 开发工具 4](#_Toc1313537953)

[2. 服务器架构 5](#_Toc500484911)

[2.1 数据存储 5](#_Toc2103937525)

[2.2 逻辑架构 6](#_Toc412251173)

[2.2.1 GameServer 6](#_Toc923219389)

[2.3 事务处理 7](#_Toc978921348)

[2.4 架构目标 8](#_Toc858876169)

[2.4.1 模块化设计 8](#_Toc1894180896)

[2.4.2 扩展性 8](#_Toc1200735944)

[2.4.3 维护性 9](#_Toc865179949)

[2.4.4 容灾 9](#_Toc467629006)

[3. 设计细节 10](#_Toc1798039469)

[3.1 设计观念 10](#_Toc259474899)

[3.2 核心服务 10](#_Toc1602824083)

[3.2.1 ConnSvr 10](#_Toc629495013)

[3.3 协程框架 11](#_Toc1418238369)

[3.4 日志 12](#_Toc1411269730)

[4. 性能 14](#_Toc253470995)

[5. 部署 15](#_Toc1626940964)

[5.1 服务器初始化 15](#_Toc87504697)

[5.2 相关环境配置 15](#_Toc1812627931)

[5.3 部署 15](#_Toc634619975)

[6. Go语言特性 16](#_Toc1654128823)

[6.1 语言特性 16](#_Toc1767317746)

[6.2 Go语言与C++的区别 16](#_Toc1463035365)

[6.2.1代码格式 16](#_Toc547621405)

[6.2.2选择语句 16](#_Toc1905526440)

[6.2.3可见性 17](#_Toc759249369)

[6.3 其他注意点 17](#_Toc356314309)

[6.3.1值传递与引用传递 17](#_Toc1390183527)

[6.3.2 Defer 18](#_Toc192458929)

[6.3.3 Go语言中不存在未初始化的变量 18](#_Toc546847321)

[6.3.4 ProtoBuf初始化时空指针问题 18](#_Toc1780398534)

[7. MainSvr 19](#_Toc121023640)

[7.1 主函数 19](#_Toc377303771)

[7.2 Bus 21](#_Toc1972753253)

[7.3 事务管理器 22](#_Toc1063897138)

[7.4 角色数据 24](#_Toc970353444)

[8. 使用的开源工具 26](#_Toc739517990)

[8.1 Redis 26](#_Toc1590992741)

[8.2 ZooKepper 26](#_Toc1496109190)

[8.3 Rabbitmq 26](#_Toc221133607)

[8.4 Logstash 26](#_Toc1445823539)

# 1. 环境

## 软件需求

* Linux （Centos7 or other linux）

## 1.2 硬件

* 服务器可以用多台机器组成集群，对单台物理服务器无特殊要求，普通Linux服务器即可

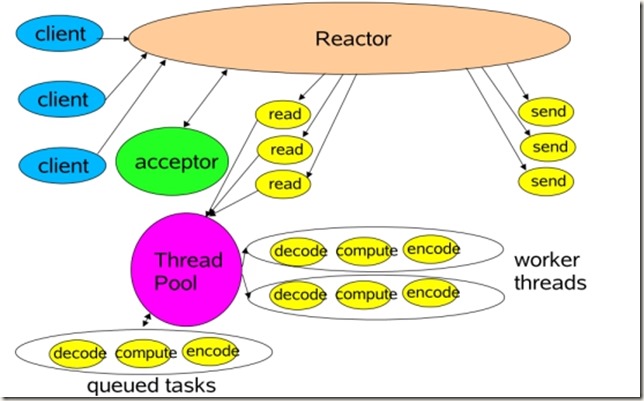
## 1.3 开发工具

* 开发语言： Golang(Shell、Python用于部署工具等开发)
* IDE：JetBrains GoLand 2018.2.1 x64
* 调试工具：IDE 单步调试、Go Test单元测试
* 版本管理工具：SVN
* 协议：ProtoBuffer 3
* 测试工具：Go Test，PProf(性能分析)
* 数据库：Mysql，Redis
* 第三方软件：ZooKeeper(配置管理), RabbitMQ(消息队列), Logstash(日志处理), Ansbile(应用部署及服务器初始化), GCM(后台服务部署),Beego(http服务支持)

# 2. 服务器架构

## 2.1 架构模型

1. **各服务进程主用** ： **Reactor** 模型 （反应器模式是一种处理一个或多个客户端并发交付服务请求的事件设计模式，当请求抵达后，服务处理程序使用I/O多路复用策略，然后同步地派发这些请求至相关的请求处理程序）



(底层网络库使用官方提供的gonet，也可以替换成开源的gnet)

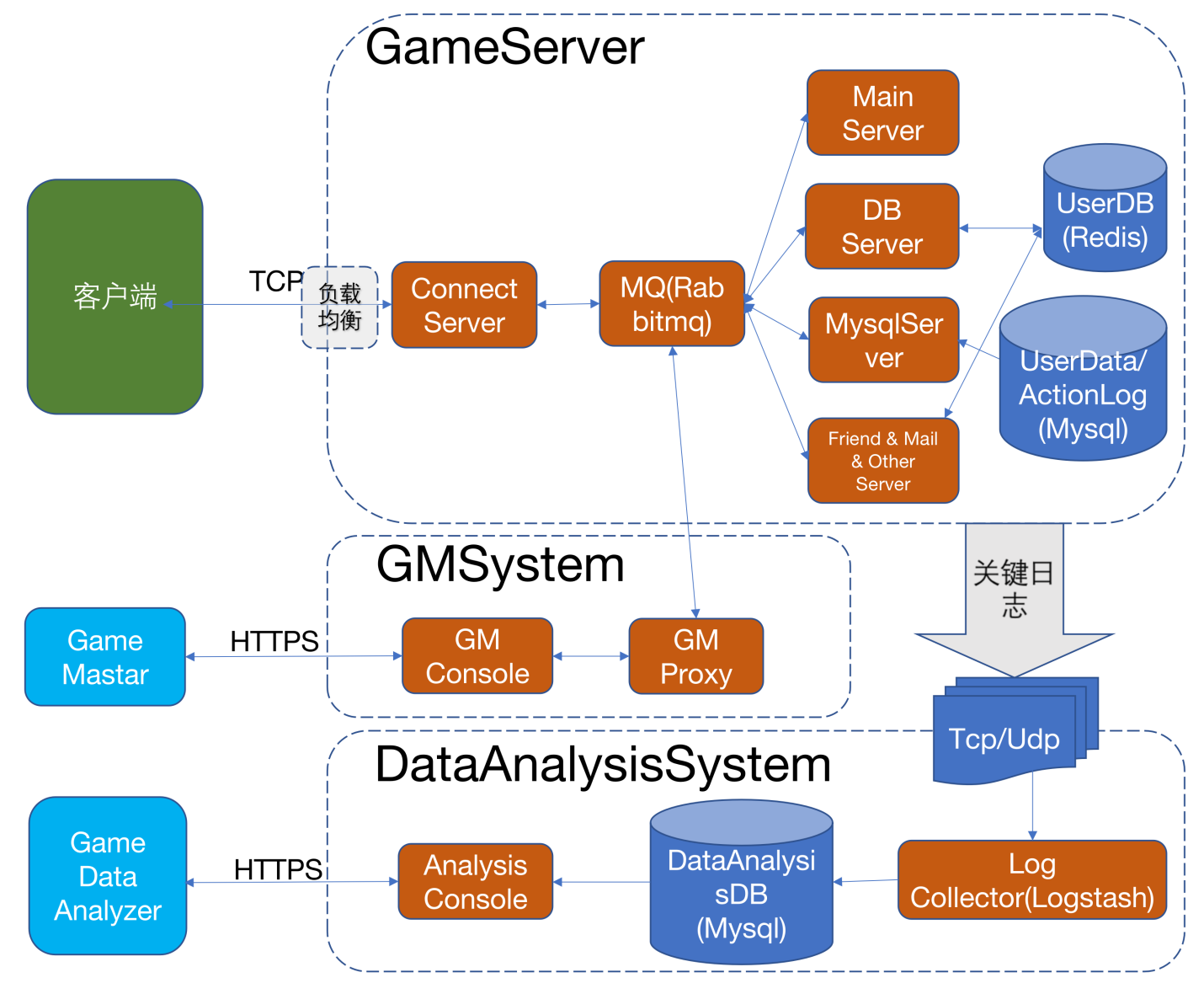
## 2.2 数据存储

数据存储有如下特性：

* 数据存储采用Redis+Mysql的模式，玩家的主要数据存储在Redis数据库中，部分玩家信息及所有的流水数据存储在Mysql中便于查找和统计，例如玩家姓名的模糊查找、DAU的统计等
* Redis数据库中的数据以[数据类型+玩家ID]为key，以ProtoBuffer序列化后的数据为值
* 典型的玩家数据的ProtoBuffer3结构如下，按照数据的类型分段：



## 2.3 逻辑架构



上图是游戏的服务器逻辑架构，主要分为三大块

* GameServer：所有的游戏服务进程集群
* GMSystem：GM管理系统
* DataAnalysisSystem：数据分析平台

### **2.3.1 GameServer**

GameServer是由游戏的逻辑进程组组成服务集群，主要包括如下进程

|  |  |  |
| --- | --- | --- |
| **进程** | **名称** | **功能** |
| **AccountSvr**  **（TODO）** | 账号服务器 | * 处理账号和鉴权 |
| **ConnSvr** | Connect Server,  连接服务器 | * 保持与玩家的TCP长连接 * 接收/发送游戏的指令包/应答包 |
| **MainSvr** | 主逻辑服务器 | * 共享内存中缓存玩家数据 * 运行绝大多数单局外逻辑 |
| **DBSvr** | 数据库服务器 | * 读/写数据库的代理 |
| **MysqlSvr** | 数据库服务器 | * 读/写数据库的代理 |
| **UserDB(Redis)** | Redis数据库 | * 存储玩家数据 |
| **UserDB(MysqlDB)** | MysqlDB数据库 | * 存储部分玩家信息、及所有的流水日志数据 |
| **MailSvr** | 邮件服务器 | * 运行邮件相关逻辑 |
| **RabbitMQ** | MessageQueue,  消息转发服务 | * 负责消息的接收和转发 |
| **ZooKeeper** | 配置管理服务 | * 负责配置管理及服务发现 |

GameServer内的进程都是Linux进程，每台物理机器都可以部署任意多个不同类型的进程，具体部署可根据实际情况灵活部署，所有进程都运行在同样的数据中心（带宽高延时低）。

## 2.4 事务处理

当客户端发送一个命令字到服务器，服务器处理这个命令字的过程称之为一个事务。

服务器处理事务的时候，以**Go协程的异步模型**进行处理，避免不必要的等待。

下面展示了一个完整的登入流程：

1. 有客户端连接到ConnSvr监听的端口时,ConnSvr会新起2个协程，分别用于消息的读和写
2. 客户端发送CMD\_MAIN\_LOGIN\_REQ命令字到ConnSvr
3. ConnSvr转发至RabbitMQ
4. RabbitMQ通过选用的路由策略转发至对应的MainSvr
5. MainSvr从内存中查找用户如果没有该用户，如果没有则从DBSvr拉取，如果数据库也没有则认为是新用户，则新建角色
6. 登入成功后，通过MainSvr->RabbitMQ->ConnSvr->客户端，给客户端回包，MainSvr处理登入后相关事务

## 2.5 架构目标

### **2.5.1 模块化设计**

由于服务器的设计目标尽量简单稳定，我们将不同的功能放到不同的服务器中实现，相对独立的模块都做成独立的服务器，在功能将尽量解耦合和模块数量之间取一个合适的平衡点。

具体实现中，我们将邮件等功能模块独立成单独的服务器进程，即便某个功能模块出现异常停止服务，也不会影响游戏的其它功能。

### **2.5.2 扩展性**

从设计层面，服务的每一类进程都支持动态平行扩展，当某类服务器负载很大的时候，我们可以不停服动态添加进程实例到服务集群当中。

同样的，如果某类进程负载很低的话，我们可以动态移除某些服务进程以节约资源的消耗。

### **2.5.3 维护性**

所有的服务进程都采用相同的程序框架，目录结构，启停脚本，启动流程，在版本更新的时候得到了很大的便利性。一套部署工具可以很方便的部署所有的进程。

当进程遇到故障需要做局部更新的时候，只需要保持ConnSvr进程一直运行，其他进程可以很方便的进行更新和重启操作，客户端无任何感知。

### **2.5.4 容灾**

所有的服务进程都可以采取多实例部署，正常情况下，可以配置两种容灾模式：

一种是主备模式，一个处理业务运转，另一个正常情况下不处理业务。当主进程异常，ZooKeeper检测不到此进程的时候，ZooKeeper会发送子节点的变化到程序，程序会将请求转发给备份进程。

另一种是负载均衡模式，例如DBSvr，请求转发给多个实例中的任何一个DBSvr是无差别的，因此在路由到DBSvr的时候可以采取的Hash值来转发。当实例数目变化时，ZooKeeper会发送子节点的变化到程序，程序会重新选择目标地址。

# 3. 设计细节

## 3.1 设计观念

GoOne的设计旨在于更容易地维护、可扩展、模块化、高可用。详情请见2.5

其他额外原则包括：

* 让游戏玩法相关的配置简单容易，无须大量程序支持，这样能够高效快速对游戏进行调整。
* 条件允许时，尽量增加代码的复用性，不写重复性代码。
* 使用合理的算法与规范，注重细节，将硬件性能发挥到最优。

## 3.2 核心服务

本节重点介绍一下后台几个核心服务器的设计实现。

### **3.2.1 ConnSvr**

ConnSvr是直接对接客户端的连接服务器，管理与客户端的TCP长连接。

在网络层面，抽象出了一个TcpServer包，其他模块如果需要使用网络功能，可以很简单的通过调用TcpServer包来实现。

通常我们在部署ConnSvr的时候，根据用户规模来评估需要部署的实例数量。客户端在连接服务器的时候，并非直接连到ConnSvr的地址，而是连到一个负载均衡器（LB），由LB将连接均衡到多个ConnSvr实例。

客户端与服务器的交互是以命令字+包体的形式，ConnSvr接到客户端指令后会转发给RabbitMQ，RabbitMQ发给相应的业务服务器（多数情况下是转发给MainSvr），然后返回处理结果到ConnSvr，最终返回给客户端。

## 3.3 协程框架

主要协程表，如下图（不包括第三方库开启的协程）：

|  |  |  |
| --- | --- | --- |
| Svr | 协程类型 | 详细的协程及功能 |
| ConnSvr | **主协程** | 主协程：处理信号及定时任务 |
| 其他 常驻协程 | pprof：一个性能测试的http服务 |
| zookeeper：负责连接zookeeper，监听子节点的变化 |
| bus：负责连接MessageQueue进行消息的收发 |
| transMgr：事务管理器负责管理所有的事务 |
| listen：负责监听TCP连接 |
| 动态协程 | reload：负责配置重新加载 |
| 消息的推送（每个客户端连接开启一个协程） |
| trans：处理收到的消息（每条消息一个协程） |
| TcpServer：负责和客户端的直接通讯（每个连接读和写共两个协程） |
| MainSvr | **主协程** | 主协程：处理信号及定时任务 |
| 其他 常驻协程 | pprof：一个性能测试的http服务 |
| zookeeper：负责连接zookeeper，监听子节点的变化 |
| bus：负责连接MessageQueue进行消息的收发 |
| transMgr：事务管理器负责管理所有的事务 |
| 动态协程 | reload：负责配置重新加载 |
| 一个定时任务：内存中删除没有心跳的用户数据 |
| trans：处理收到的消息（每条消息一个协程） |

程序启动后，主协程对使用到的各种资源进行初始化，主要工作包括但不限于：

* 初始化clog
* 初始化ZooKeeper和配置
* 初始化Bus
* 初始化信号处理
* 初始化事务

初始化完成后，主协程循环接收及处理信号和定时任务。

Bus协程连接RabbitMQ后收取指令包后交给事务协程处理。处理完后再经RabbitMQ发送。

## 3.4 日志

游戏有两种日志格式：本地日志和网络日志。

本地日志写入本地磁盘，文件以小时为单位切分，同时会设置一个文件最大大小，文件超过了这个大小也会切换文件。

每个业务进程都调用相同的日志API来写日志，写日志是一个单独的线程，从而能最大程度的减少写日志对业务的影响。

本地日志分为5个等级：debug，info，warn，error，fatal。通过配置可以控制进程开启的日志等级。一般来说测试环境开启全部，而正式环境一般开启info，warn，error和fatal四个等级。对于error级别的日志，需要进行监控，这个级别日志如果出现，代表业务进程有异常，需要进行人工排查。

典型的日志如下所示，有日志时间，进程ID，产生日志的文件行数，和自定义的日志内容。



网络日志即前面提及的关键日志，通常是玩家的操作，道具流水等一些重要的日志，需要记录这类型日志需要采用一定的格式通过TCP发送给LogCollector，最终会写入到MysqlDB数据库中。

# 4. 性能

服务器进程的瓶颈主要在CPU上，在普通的单核2.2GHz频率的CPU下：

* ConnSvr一个实例能维持20000个客户端连接(受限于自身设置的连接数上限)
* MainSvr一个实例每秒钟能处理1500个登录事务
* 其它业务服务器由于业务较轻负载均比较低

# 部署

服务器采用Ansible工具部署。所有部署相关文件都在工程的deploy文件夹，本节提到的目录均以deploy为根目录

## 5.1 服务器初始化

使用Ansible脚本inithost/inithost.yml来初始化服务器

## 5.2 相关环境配置

* 配置机器：

在host目录下保存机器的配置

* 配置部署结构

在playbook\_dev文件夹下添加两个文件：

1. 机器上安装的实例文件，类似于dev1.yml

2. 环境对应的参数，类似dev1\_vars

* 配置实例

在roles文件夹下添加实例，已经实例对应的操作，可以参考roles/mainsvr的配置

## 5.3 部署

用脚本deploy.sh进行部署操作，相关操作方法可以参考文件注释

## 5.4 使用docker配置来快速部署服务器（后续补充）

# 6. Go语言特性

## 6.1 语言特性

* 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

## 6.2 Go语言与C++的区别

### **6.2.1代码格式**

Go语言要求代码的花括号{}必须采用统一的风格，即左花括号{必须放在行尾而不能另起一行。

Go不需要在语句的最后增加分号;以表示语句结束。Go循环/判断语句无需加()。

### **6.2.2选择语句**

Go的选择语句中，不支持break关键字，因为Go的选择语句中，遇到符合条件的分支会自动跳出；如果想要继续向下执行，需要在case分支中增加fallthrough关键字。

### **6.2.3可见性**

Go语言中没有private/protected/public/friend关键字，可见性是通过成员的首字母是否大写决定的，并且仅支持包间的可见性定义，不支持类的可见性。

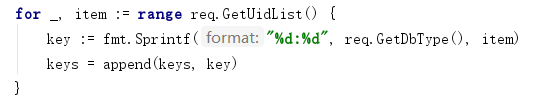
包中所有首字母大写开头的成员（对象、函数），表示包间可见，可以被其它包中的代码访问。但对于首字母小写开头的成员，则仅能被本包中的成员所访问（这相当于包内的所有成员间都加了friend声明）

## 6.3 其他注意点

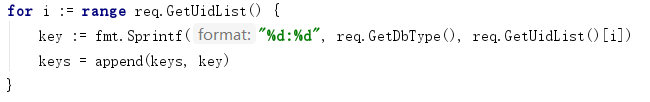
### **6.3.1值传递与引用传递**

Go语言中所有的传参都是值传递（传值），都是一个副本，一个拷贝。因为拷贝的内容有时候是非引用类型（int、string、struct、数组等这些），这样就在函数中就无法修改原内容数据；有的是引用类型（指针、map、slice、chan等这些），这样就可以修改原内容数据。注意：数组是非引用类型，这点和C++不同。如果要修改原内容数据，或者数据内容过大，考虑性能应该用切片或指针（一般用切片）。

注意:下面代码item也是值传递，不能修改原内容数据。



如果要修改原内容数据，或者数据内容过大，考虑性能应该用



### **6.3.2 Defer**

多个defer出现的时候，多个defer之间按照LIFO（后进先出）的顺序执行

### **6.3.3 Go语言中不存在未初始化的变量**

* 数值变量对应的是0值
* 布尔变量对应的是false
* 字符串对应的零值是空字符串
* 接口或者引用类型（包括slice，map，chan）变量对应的是nil
* 数组或者结构体等聚合类型对应的零值是每个元素或字段对应该类型的零值。

### **6.3.4 ProtoBuf初始化时空指针问题**

目前使用的Go版ProtoBuf（版本号：2.6.1）所有成员、基础类型都是指针，所以在初始化时一定要注意判断空指针。

请用GetXXX方法作为读方法。虽然结果和直接读该成员相同，但当该成员为空时（或者其上一级为空）能返回0值，避免服务器崩溃。

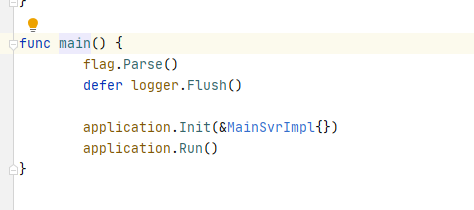
写方法，只能自己初始化。尤其是嵌套结构每一层都需要初始化。

# 7. MainSvr

因为MainSvr缓存有玩家角色数据，所以MainSvr担负处理绝大多数的玩家请求。其他Svr如果需要用到（读/写）玩家数据，则需要同MainSvr交互完成。因此，MainSvr至关重要，下面将以MainSvr为例详解Svr架构。

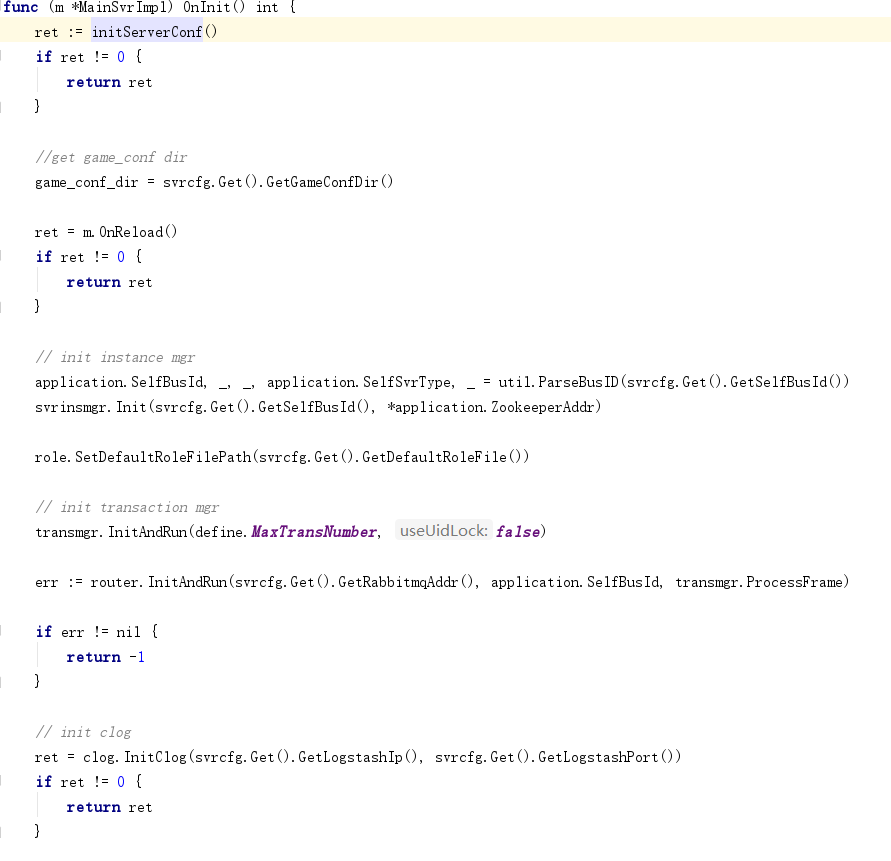
## 7.1 主函数

主函数，如下图：



所有的Svr主函数都类似，包含：**解析命令行参数**、**pprof性能实测**、**初始化**、**运行主协程**。

其中MainSvr的初始化如下图：



初始化做了以下工作：

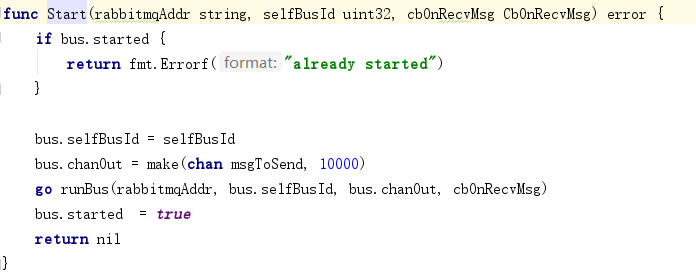
1. 读取配置
2. 运行一次Reload
3. 初始化并运行ZooKeeper
4. 设置默认角色数据
5. 初始化并运行trans
6. 初始化并运行router
7. 初始化并运行clog

其中1,3,5,6为所有Svr都必须做的。

而3,5,6会新开一协程运行。

## 7.2 Bus

初始化时会设置好自己的BusId，并另起一协程运行Bus：



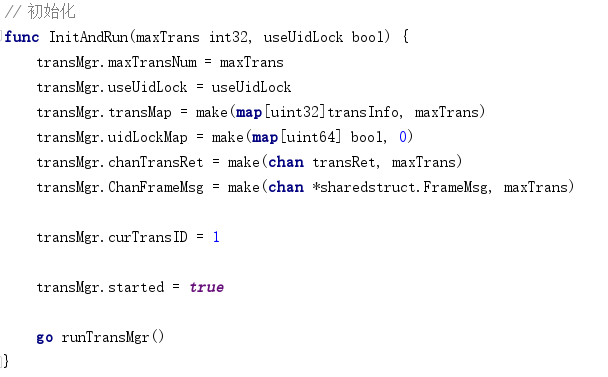
Bus运行时，一直循环接受通道里的内容进行收发消息，收到的消息交由事务管理器去处理。

底层是通过连接MQ来完成收发消息的工作。



## 7.3 事务管理器

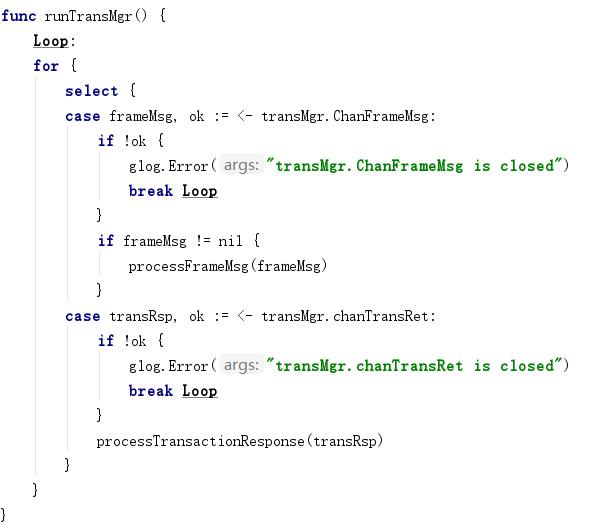
事务管理器同样会另起一协程运行：



事务管理器运行时，一直循环接受2个通道的消息。

第一个通道接受新的消息，收到后会新起一协程处理。

另一通道接受处理完的消息，收到后删除这个事务。

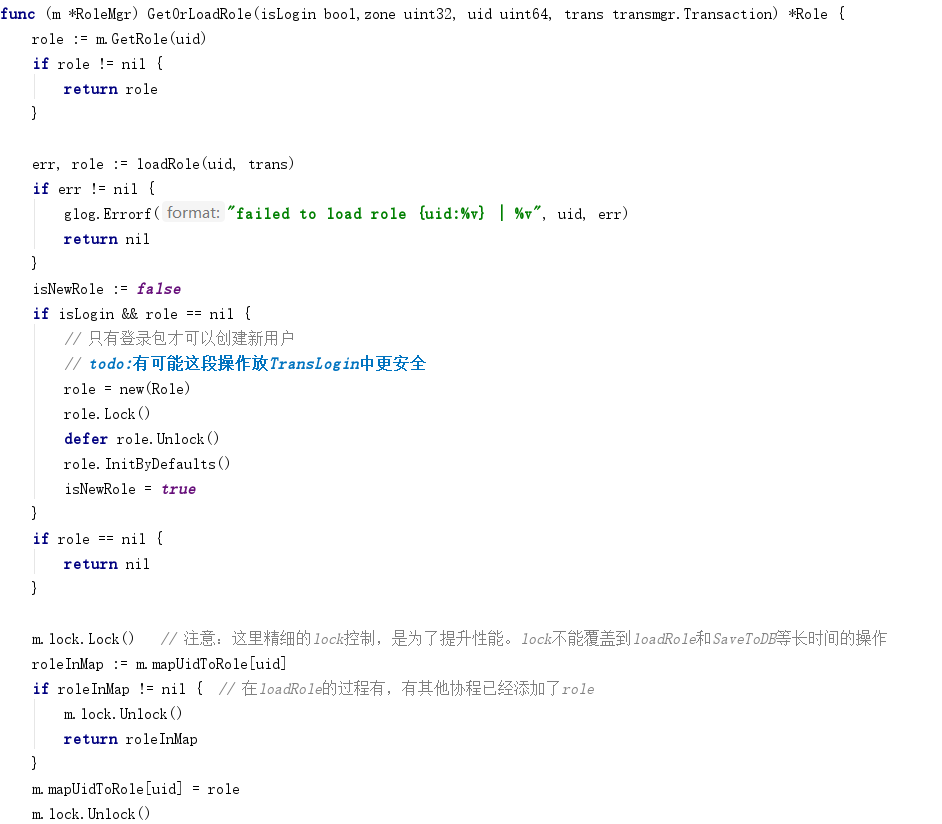


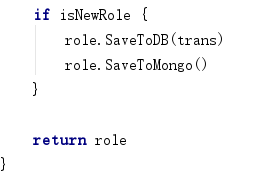
## 7.4 角色数据

MainSvr事务处理需要用到role的数据，由于协程的安全性与代码的简洁性考虑，会在事务处理前统一获取role的数据。

而role的数据是由roleMgr统一管理。

先从内存中取，如果没有则去数据库拉取，如果还是没有同事是登入包则可以创建一个新的角色数据。





# 使用的开源工具

## 8.1 Redis

* 速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)
* 支持丰富数据类型，支持string，list，set，sorted set，hash
* 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行
* 丰富的特性：可用于缓存，消息，按key设置过期时间，过期后将会自动删除

## 8.2 ZooKepper

* 分布式配置服务
* 简单易用的接口

## 8.3 Rabbitmq

* 消息队列
* 支持集群
* 消息持久化
* 削峰

## 8.4 Logstash

* 开源的服务端数据处理管道(日志系统)
* 可以从多个来源采集数据，转换数据，然后将数据发送到不同的 “存储库”