动态覆盖率方案探讨（new）

- 背景
- 代码覆盖率的意义
- 什么是动态覆盖率
- 业界解决方案
- 聊聊golang插桩技术
- 项目插桩原理分析
    - 项目插桩流程概览
    - 源码插桩详细过程
- 整体业务流程
    - 业务流程介绍
    - 部署流程变更
- roadmap
- 参考资料

背景

目前credit业务迅速发展，随着业务量的增长、多国家的快速部署，代码质量的保证已经变得越来越重要。为了提高代码质量，credit业务已经引入了静态代码测试覆盖率工具：go test+sonar。然而对于项目的集成测试，依然无能为力。

本文主要介绍一种实现动态代码覆盖率的方案探讨，主要包含如下内容：

1.什么是动态覆盖率

2. 代码覆盖率的意义

3.业界解决方案

4.聊聊golang插桩技术

5.项目插桩原理分析

6.项目业务流程

7.roadmap

8.参考资料

代码覆盖率的意义

1. 分析未覆盖部分的代码，从而反推在前期测试设计是否充分，没有覆盖到的代码是否是测试设计的盲点，为什么没有考虑到？需求/设计不够清晰，测试设计的理解有误，工程方法应用后的造成的策略性放弃等等，之后进行补充测试用例设计。
2. 检测出程序中的废代码，可以逆向反推在代码设计中思维混乱点，提醒设计/开发人员理清代码逻辑关系，提升代码质量。
3. 代码覆盖率高不能说明代码质量高，但是反过来看，代码覆盖率低，代码质量不会高到哪里去，可以作为测试自我审视的重要工具之一。

什么是动态覆盖率

动态代码覆盖率是相对静态代码覆盖率而言的。静态代码覆盖率的收集需要停机才能收集，比如使用go test输出测试覆盖率。动态代码覆盖率是指在项目的运行中实时收集覆盖率。和静态代码覆盖率不同，动态覆盖率的统计不需要研发编写测试用例，而是通过集成测试收集的，如研发test环境验证通过app或者接口验证自身功能的正确性，或者QA跑功能测试用例等。

业界解决方案

	Jacoco	Emma	Cobertura	gcov	go tool cover	goc

原理	使用asm修改字节码	可以修改Jar文件、class文件	基于Jcoverage，基于ASM框架对class插桩	编译阶段插桩	源码插桩	源码插桩
覆盖粒度	类、方法、行、分支、指令、圈	类、方法、行、块	行、分支	行、分支	行、分支	行、分支
插桩	on-the-fly和offline	on-the-fly和offline	offline	编译阶段插桩	源码插桩	源码插桩
性能	快	较快	较快			
缺点		不支持JDK8	关闭服务器才能获取覆盖率报告	关闭服务器才能获取覆盖率报告	关闭服务器才能获取覆盖率报告	无法集成Sonar、无增量覆盖率、无管理平台
支持语言	Java	Java	Java	c	golang	golang

目前针对动态覆盖率的探讨主要有以下3种方式：

源码插桩：此种方式会修改源码，在源码中动态增加覆盖率统计变量，如golang中的go tool cover,goc等

编译阶段插桩：此种方式是在编译时，生成汇编文件时插桩，如gcov

字节码插桩：此种方式主要属于JAVA系，通过动态修改运行时的字节码或者修改class文件的方式插桩，如jacoco

其中针对go语言，目前的解决方案都是通过源码插桩的方式实现的。

那么如果直接使用go tool cover工具，能否直接完成项目的插桩的，答案是否定的，主要存在如下问题：

1. 程序必须关闭才能收集覆盖率
2. 受限于 `go test-c` 命令的先天缺陷，它会给被测程序注入一些测试专属的 flag，比如 -test.coverprofile, -test.timeout 等等，会破坏被测程序的启动姿势
3. go tool cover只能逐个文件添加统计量

基于以上原因，go tool工具并不适合我们的需求。

goc项目是目前go语言探讨动态覆盖率的一个开源项目，本项目计划也是在goc项目上改造，

目前goc项目依然存在如下问题：

（1）无法和CI/CD流程继承，如gitlab，sonar、jenkins等

（2）缺少一个统一的管理平台

（3）无法在运行时清空统计量

（4）无法实现增量覆盖率

（5）源码插桩时间较长

聊聊golang插桩技术

go语言采用的是插桩源码的形式，而不是待二进制执行时在设置breakpoints，这就导致了go的测试覆盖率收集，一定是侵入式的，会修改目标程序的源码。因此，建议插过桩的代码，不要放到线上（代码预估增长10%）。

那么到底什么是插桩的？为此，我们假设有如下一段代码：

```
1  package main
2
3  import "fmt"
4
5  func Print() {
6     fmt.Printf("test")
7  }
8  func main() {
9     fmt.Println("Hello")
10    Print()
11 }
```

```
go tool cover -mode=count -var=CoverageVariableName main.go
```

```
package main

import "fmt"

func Print() {CoverageVariableName.Count[0]++;
        fmt.Printf("test")
}
func main() {CoverageVariableName.Count[1]++;
        fmt.Println("Hello")
        Print()
}

var CoverageVariableName = struct {
        Count     [2]uint32
        Pos       [3 * 2]uint32
        NumStmt   [2]uint16
} {
        Pos: [3 * 2]uint32{
                5, 7, 0x2000e, // [0]
                8, 11, 0x2000d, // [1]
        },
        NumStmt: [2]uint16{
                1, // 0
                2, // 1
        },
}
```

可以看到，执行完之后，源码里多了个CoverageVariableName变量，其有三个比较关键的属性:

- Count uint32 数组，数组中每个元素代表相应基本块 (basic block) 被执行到的次数
- Pos 代表的各个基本块在源码文件中的位置，三个为一组。比如这里的5代表该基本块的起始行数，7代表结束行数,0x2000e比较有趣，其前 16 位代表结束列数，后 16 位代表起始列数。通过行和列能唯一确定一个点，而通过起始点和结束点，就能精确表达某基本块在源码文件中的物理范围
- NumStmt 代表相应基本块范围内有多少语句 (statement)

CoverageVariableName变量会在每个执行逻辑单元设置个计数器，比如CoverageVariableName.Count[0]++, 而这就是所谓插桩了。通过这个计数器能很方便的计算出这块代码是否被执行到，以及执行了多少次。相信大家一定见过表示 go 覆盖率结果的 coverprofile 数据，类似下面:
`github.com/gococo/main.go:21.13,23.2 1 1`

这里的内容就是通过类似上面的变量CoverageVariableName得到。其基本语义为
"**文件:起始行.起始列,结束行.结束列 该基本块中的语句数量 该基本块被执行到的次数**"

项目插桩原理分析

项目插桩流程概览

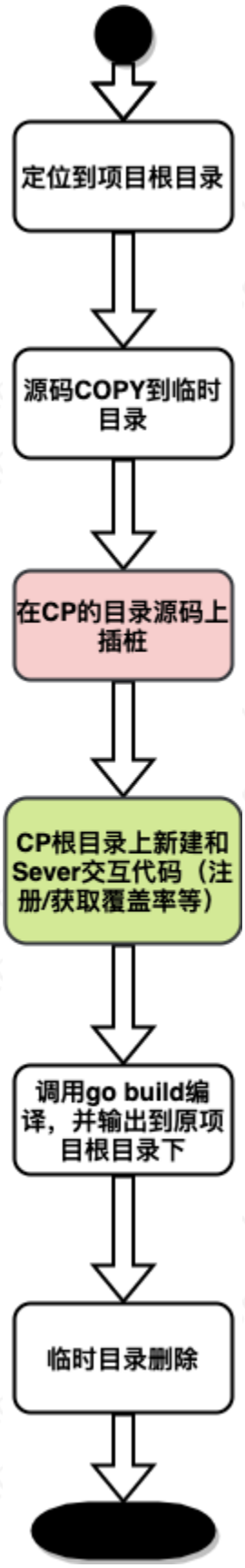


图1 项目插桩原理概览 https://app.diagrams.net/#G1QnUzMOaipSu-KtKajbiCrv771hATgUce

插桩过程大致分成如图的步骤：

步骤1: 定位到项目的根目录

执行此步骤，主要是需要使用go list命令列举包几文件，这样我们知道要复制那些文件。同时针对go项目，也区分为

标准项目：使用go.mod项目，

遗留项目，是指go 1.11，go 1.12项目，此类项目找不到Build.root

非标准项目：没有go.mod的项目

步骤2: 源码Copy到临时目录

步骤3: 在Copy的临时目录上进行源码插桩

此步骤是插桩的核心步骤，比较复杂。会独立一个流程讲解

步骤4: 在临时目录上注入一段和server交互的代码，主要用于自身项目的注册，及对外提供生成覆盖率的接口

步骤5: 调用go build对插桩后的代码进行编译，并把编译的目标文件输出到原项目目录的根目录下

步骤6: 清理工作，删除新建的临时目录

源码插桩详细过程

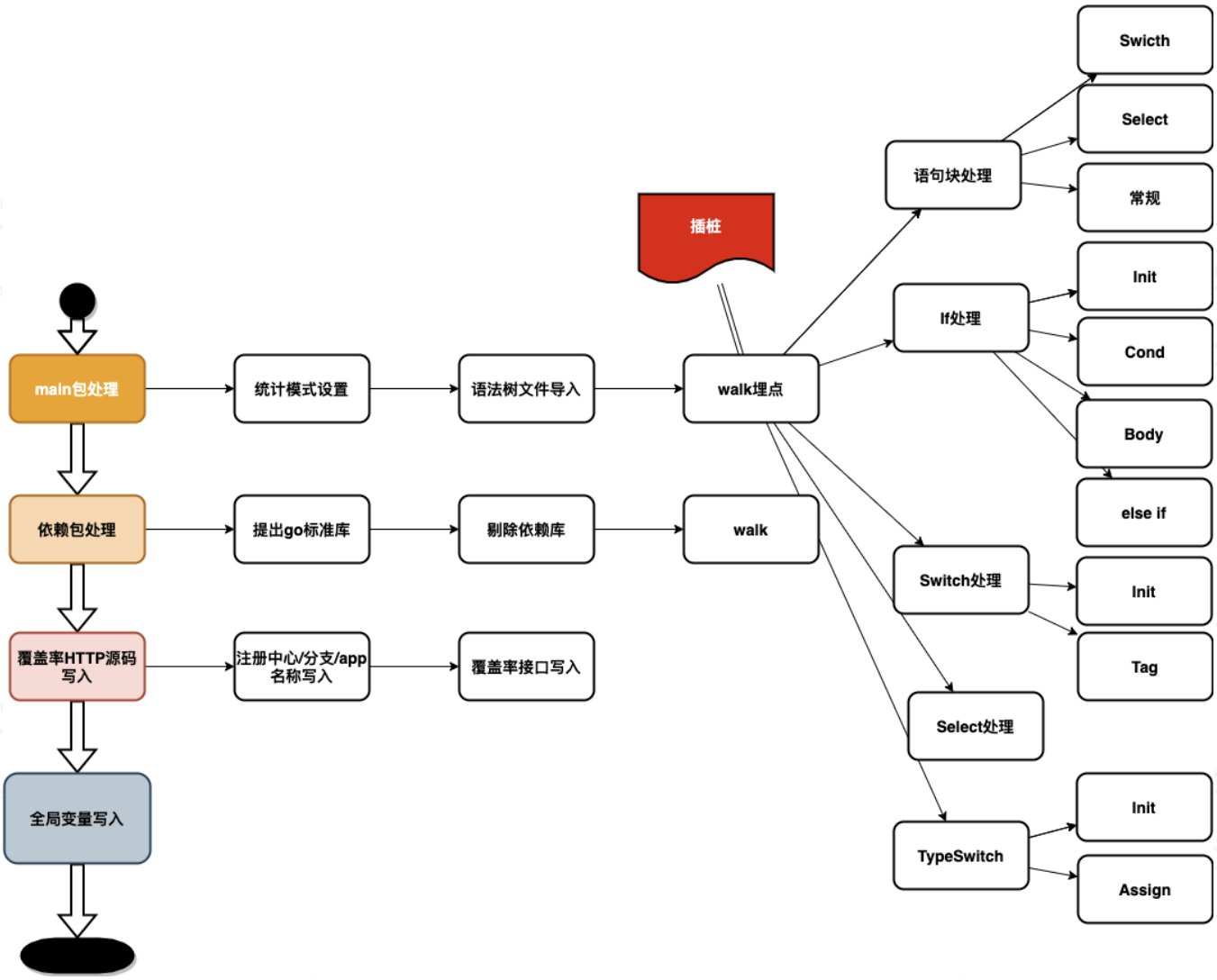


图2 源码插桩详细流程 https://app.diagrams.net/#G1mzTwD5eqWLuN2tYtZaa6bh5NNV1YdkCo

源码插桩，简单来说，是从main包开始的一个递归调用过程。具体来说如下：

（1）加载某个源文件

（2）使用golang语法树解析

（3）对Block块语句，IF语句，Select语句，TypeSwicth语句进行解析。

什么是语句块：类似 { doing()}的语句，语句块本身也是递归的。

（4）插入埋点数据（类似上述案例：CoverageVariableName.Count[0]++ ）

（5）注入和Server交互的代码（单独创建一个文件）

（6）把生成的全局变量，统一输出到一个新文件中，以便其他包引用

整体业务流程

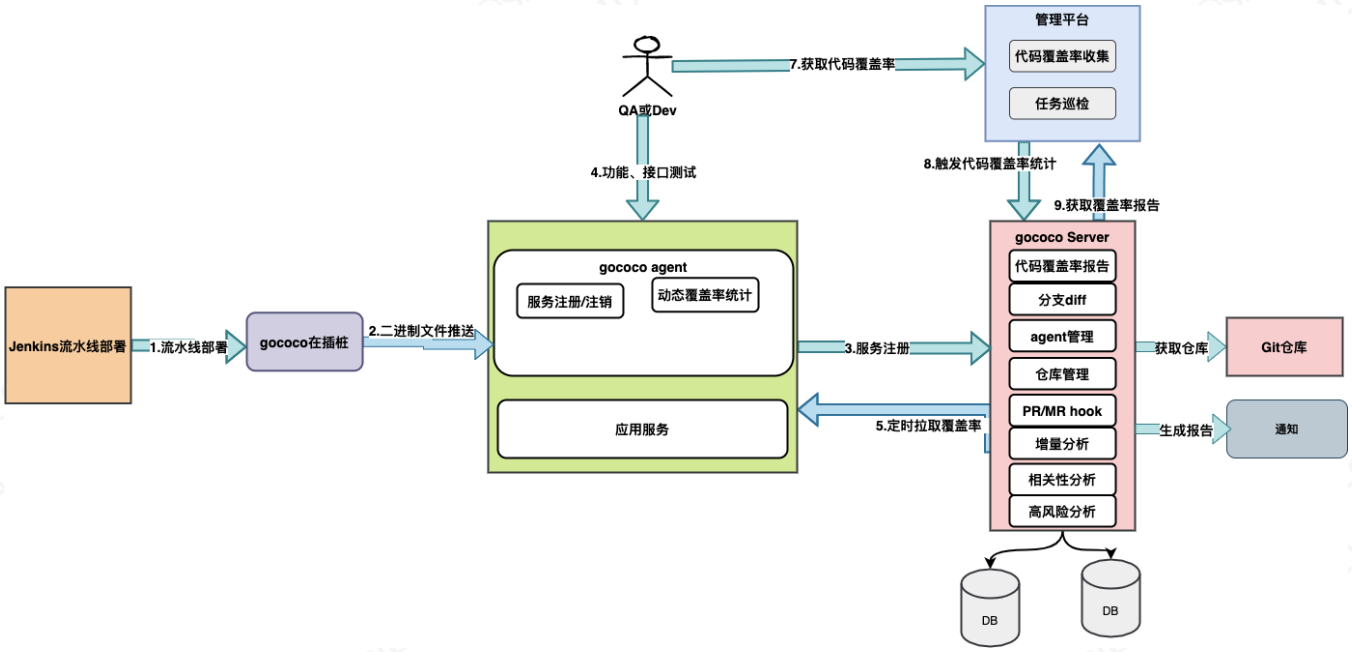


图3 整体业务流程https://app.diagrams.net/#G1CmqmRpCpPVwyyheKBPOoGL5fqHa6pfBg

业务流程介绍

基本的流程如图3:

步骤1: 通过jenkins流水线部署项目，此处部署时需要指定编译的脚本（备注：建议test环境和live环境编译脚本区分开）

步骤2: 编译脚本调用我们的agent,对源码进行插桩

步骤3: 编译后的目标文件推送到服务器，并启动

步骤4: 启动后的脚本，主动向server进行注册，并和server保持心跳

步骤5: QA或者Dev进行功能、接口测试

步骤6: server定时向agent拉取当前的覆盖率

步骤7: sonar-scanner上传覆盖率报告（是否可以使用API操作，还需详细了解）

步骤8: QA或者Dev获取当前的代码覆盖率，并生成报告

部署流程变更

（1）业务打包脚本修改

编译脚本新增（分支名称，commit_log，环境信息）参数

针对非Test环境，调用go build编译（保持不变）

针对Test环境，调用gococo agent进行编译（插桩并go build编译）

（2）修改Jenkins Shell命令，增加（分支名称，commit_log,环境信息）

（3）Jenkins所在服务器推送gococo二进制文件

针对物理服务器，直接推送就可以了

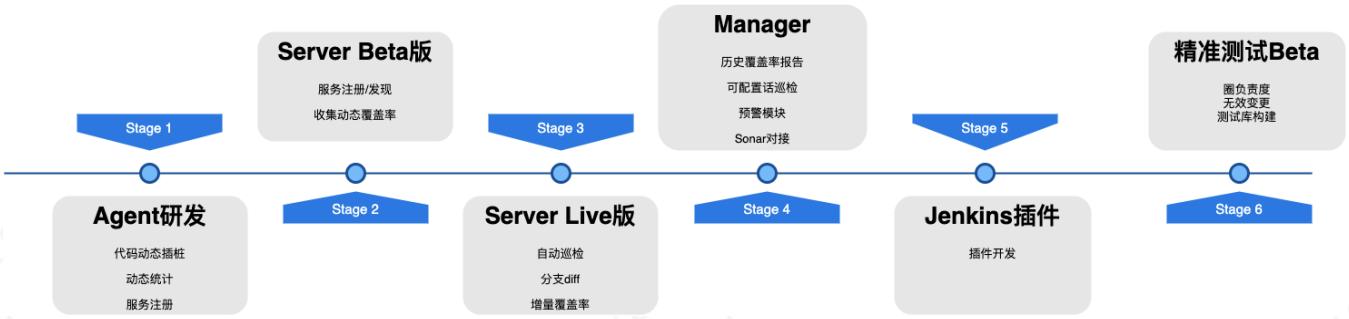
针对k8s，可以通过共享目录或者docker cp或者应用代码库直接推送

也可以开发jenkins插件

（4）测试机所在服务器部署sonnar-scanner（待深入调研）

sonnar-scanner主要用于测试报告上传

roadmap



gococo roadmaphttps://app.diagrams.net/?libs=general#G1hL87HC24GdSpB-giKIfqLmPs2oOyQ8_L

阶段	Stage1	Stage2	Stage3	Stage4	Stage5	Stage6
内容	Agent研发	Server Beata版	Server Live版	Manager	Jenkins插件	精准化测试-Beta
时间	2021.4-2021.5	2021.5-2021.6	2021.6-2021.7	2021.7-2021.8	2021.8-2021.9	2021.9-2021.10

参考资料

goc git地址：https://github.com/qiniu/goc

谈谈代码覆盖率： https://tech.youzan.com/code-coverage/

有赞代码覆盖率的探讨：https://www.infoq.cn/article/TBLtJ8a6PQ3X1uQIWlXo