

开发记录

卷二

Dolphin

Copyright © 2017 Xiaoqiang Jiang

EDITED BY XIAOQIANG JIANG

[HTTP://JIANGXIAOQIANG.GITHUB.COM/](http://jiangxiaoqiang.github.com/)

All Rights Reserved.

Version 23:23, January 22, 2019

Contents

I	第一部	1
1	第一章	3
1.1	常见问题排查及解决	3
1.1.1	yum 工具	3
1.1.2	Linux 版本查看	4
1.1.3	Windows 同时连接内外网	4
1.1.4	U 盘只读 (Read-only file system)	5
1.1.5	RestTemplate read time out	6
1.1.6	自动化部署 (Auto Deploy)	6
1.1.7	Nginx 超时转发	14
1.1.8	无法获取数据库连接	15
1.1.9	Java heap space	18

1.1.10	报表在现场演示环境无法打开	21
1.1.11	查询变慢	22
1.1.12	资源分配约定	23
1.1.13	Tomcat 请求头设置	23
1.1.14	GoAccess	24
1.1.15	Nginx 启用压缩	25
1.1.16	查看网站并发	25
1.1.17	自动构建更新的问题	26
1.1.18	Non-resolvable parent POM for Could not find artifact	27
1.1.19	LocalDateTime 格式化	28
1.1.20	invalid constant type: 18	30
1.1.21	视频解析	31
1.2	编辑中	31
1.2.1	缓存雪崩 (Cache stampede)	31
1.2.2	TCPCopy	31
1.2.3	Could not get a resource from the pool	32
1.2.4	服务限流	32
1.2.5	Python 绘图	32
1.2.6	XX-Net	33
1.2.7	fluxion	33
2	DB	35
2.1	Redis	35
2.1.1	Redis 消息队列	35
2.2	SSDB	36
2.3	Postgres	36
2.3.1	CTID	38
2.3.2	常用 SQL	38
2.3.3	WAL(Write-Ahead Logging)	40
2.3.4	跨库查询	40
2.3.5	统计信息	40
2.3.6	异步流复制 (Async Stream Replica)	41

3	Python	43
3.1	Python 基础 (Python Foundation)	43
3.1.1	Python 命名规范	43
3.1.2	为什么要使用类	43
3.1.3	str 与 bytes	45
3.1.4	序列化和反序列化	45
3.1.5	反反爬	45
3.1.6	调试	46
3.1.7	Unable to write settings	48
3.1.8	pipenv	48
3.1.9	Scrapy	48
3.1.10	常见问题	50
3.1.11	Google Spider	51
3.1.12	api	52
3.2	Django	54
3.2.1	使用事务 (Using Transaction)	54
3.3	日志中心 (Logging Center)	54
3.4	配置管理中心 (Configuration Management Center)	55
4	MQ(Message Queue)	63
4.1	Apache Kafka	63
4.1.1	Producer	63
4.1.2	Consumer	65
II	Tool	67
5	Widgets	69
5.1	Little Tool	69
5.1.1	traceroute	69
5.1.2	dig	70

5.1.3	ssh	71
5.2	Develop	71
5.2.1	Mybatis Aotogenerate	71
5.2.2	Medis	71
5.2.3	Consul	72
5.3	WireGuard	72
5.3.1	WireGuard 配置	72
5.3.2	使用问题	74
5.3.3	可以连接，无法上网	74
5.3.4	应用	75

6 Docker 79

6.1	Docker 基础	79
6.1.1	为什么要用 Docker	79
6.1.2	Docker 安装	80
6.1.3	安装数据层	80
6.1.4	Docker 导入导出	81

III Network 83

7 HTTP 85

7.1	Fiddler	85
7.1.1	排查 Google Api 拒绝原因	85
7.1.2	Fiddler 与 Shadowsocks 冲突	86
7.2	Web Server	86
7.2.1	CIDR(Classless Inter-Domain Routing)	86
7.2.2	maxPostSize	87
7.2.3	URL 自动重定向	88

IV	Book	89
8	Paper	91
8.1	Font	91
8.1.1	类型	91
8.2	纸的种类	91
8.2.1	宣纸 (Xuan Paper)	91
8.2.2	连史纸	92
8.2.3	美浓和纸	92
	Bibliography	93
	Books	93
	Articles	93

这里记录的是一些比较杂乱的笔记，绝大多数文字皆来源于网络，不是自己的原创，这里没有高深的算法，没有宏伟的技术及系统架构，只是一些平时工作中遇到的一些问题，和解决问题的思路以及所采用的方案。由于平时工作时还没有遇到前人没有遇到过的问题需要自己发明方去解决 (其实真的遇到估计也是没辙)，所以绝大部分内容是为了避免再遇到同样的问题时，又需要到处去搜寻，索性将之记录下来，以便于下次可以快刀斩乱麻，迅速解决问题。

第一部

1	第一章	3
1.1	常见问题排查及解决	
1.2	编辑中	
2	DB	35
2.1	Redis	
2.2	SSDB	
2.3	Postgres	
3	Python	43
3.1	Python 基础 (Python Foundation)	
3.2	Django	
3.3	日志中心 (Logging Center)	
3.4	配置管理中心 (Configuration Management Center)	
4	MQ(Message Queue)	63
4.1	Apache Kafka	

1. 第一章

1.1 常见问题排查及解决

1.1.1 yum 工具

在使用 yum 命令时，输出如下错误：

```
1 Could not find platform independent libraries <prefix>
2 Could not find platform dependent libraries <exec_prefix>
3 Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]
```

原来是 yum(Yellowdog Updater, Modified) 工具是用 Python 编写¹，最近删除了系统默认的 Python2.x 文件夹，破坏了 python 的默认环境。由于编写的爬虫应用默认使用 Python3，使用如下命令将 Python 指向 Python3：

```
1 # 设置命令别名，只向新版 Python
```

¹[https://en.wikipedia.org/wiki/Yum_\(software\)](https://en.wikipedia.org/wiki/Yum_(software))

```
2 alias python='/usr/bin/python3.6'
3 # 查看 Python 版本
4 python --version
```

设置 Python 的相关路径:

```
1 export PYTHONHOME=/usr/lib64/python3.6
2 export PYTHONPATH=/usr/lib64/python3.6/site-packages/
```

1.1.2 Linux 版本查看

```
1 cat /proc/version
```

Linux 内核提供了一种通过 /proc 文件系统, 在运行时访问内核内部数据结构、改变内核设置的机制。proc 文件系统是一个伪文件系统, 它只存在内存当中, 而不占用外存空间。

1.1.3 Windows 同时连接内外网

如下命令让所有流量都走 10.55.10.1:

```
1 # 添加路由语法
2 route ADD destination_network MASK subnet_mask gateway_ip
   metric_cost
3 # 添加默认路由
4 route add -p 0.0.0.0 mask 0.0.0.0 10.55.10.1
5 # 10.55 开头的流量走网关 10.55.10.1
6 route -p add 10.55.0.0 mask 255.255.0.0 10.55.10.1 metric 1
```

如果没有指定子网掩码 (Subnet Mask), 默认使用 255.255.255.0。跳数 (Metric) 数字越小, 优先级越高, Metric 为路由术语, 译为度, 指在路由选择协议算法完成计算后得到的一个变量值, 它的目的是确定最佳路由。

1.1.4 U 盘只读 (Read-only file system)

在拷贝文件到 U 盘时，提示如下：

```
1 cp: cannot create regular file '/run/media/dolphin/Fedora-WS-Live-28-1-1/note': Read-only file system
```

回想了下，应该是不久前 U 盘做了 Fedora 的刻录盘，刻录工将 U 盘设置为读保护模式，此时可以使用如下命令将 U 盘设置为读写模式：

```
1 # 设置设备/dev/sdb1 为读写模式
2 sudo blockdev --setrw /dev/sdb1
3 # 设置设备/dev/sdb1 为只读模式
4 sudo blockdev --setro /dev/sdb1
```

blockdev 命令可以获取 Linux 下块设备的属性值，以及设置一些块设备的属性值。`/dev/sdb1` 是文件系统名字。设置 U 盘为读写模式后可顺利拷贝文件到 U 盘。比较诡异的是，在运行了 block 命令后，是可以拷贝文件。但是后来不知何原因，U 盘又变成只读了，而且 blockdev 命令只读设置不再有效。后面的解决方式就是将 U 盘重新分区²，做以下操作之前提前备份文件。列出当前系统上的所有分区信息：

```
1 fdisk -l
```

在列出的分区表中，找到 U 盘的设备名字，这里是 `/dev/sdb1`。输入如下命令准备编辑分区信息：

```
1 fdisk /dev/sdb1
```

根据提示，删除旧分区信息，创建新分区信息即可。输入 `d(delete)` 删除旧分区，输入 `n(new)` 创建新分区，输入 `w(write)` 保存新分区。使用如下命令创建 FAT 文件系统：

```
1 # sdX - "SCSI" hard disk.
```

²参考 StackOverflow 的解方法：<https://unix.stackexchange.com/questions/216152/usb-disk-read-only-cannot-format-turn-off-write-protection>

```
2 # Also includes SATA and SAS.  
3 # And IDE disks using libata  
4 # (on any recent distro).  
5 mkfs.vfat -F 32 /dev/sdb1
```

1.1.5 RestTemplate read time out

运行一段时间后,系统翻页,偶尔返回空数据。经检查,服务端在使用 RestTemplate 调用接口时,偶尔会返回 Read time out 错误。Spring RestTemplate 的 Read time out 默认超时时间是 1 秒,如图3.5所示。当接口的响应时间超过 1 秒时,会出现 read time out 错误。

上图是 Debug 跟踪时查看的 RestTemplate 默认变量。由图中可以看出,默认超时时间是 1 秒。此时需要增大 Read time out 的默认超时时间。自定义 Read time out 超时时间如下代码片段所示:

```
1 @Bean(name = "commonRestTemplate")  
2 public RestTemplate restTemplate(RestTemplateBuilder  
    restTemplateBuilder) {  
3     return restTemplateBuilder.setConnectTimeout(5000)  
4         .setReadTimeout(20000)  
5         .build();  
6 }
```

出现此问题的原因是:当数据量较少时,请求基本能在非常短时间内响应,当数据量逐渐增加时,部分请求处理时间偶尔超过 1 秒,会出现偶尔失败的情况。

1.1.6 自动化部署 (Auto Deploy)

整体概况

目前已经将自动化部署应用到部分项目中,省去了非常多手工操作。目前应用的项目情况一览表,表中,CI 表示 Continuous Integration。

自动化部署服务器信息如表1.2所示。提交代码后,构建服务器定时检查代码更新,检查周期可以通过 cron 表达式定义,若有变更,则触发自动编译构建。应用构建并打包完毕后,会通过调用文件中转服务,将编译完成的应用包拷贝到测试服务器指

Table 1.1: 自动部署项目信息

项目名称	中文名	说明
report	通用数据录入子系统	通过可灵活调整的配置, 动态快速应对不同维度的数据录入场景
report-frontend	通用数据录入子系统 (前端)	通过可灵活调整的配置, 动态快速应对不同维度的数据录入场景, 提供动态渲染的, 统一的 UI
system	主应用	主应用
system-frontend	主应用 (前端)	主应用 (前端)
system-exchange	数据交换 (exchange) 子系统	应对所有数据交换场景
system-api	接口 (api) 应用	接口应用, 提供外部数据服务
message	消息系统	站内通知
message-frontend	消息系统 (前端)	站内通知
web-ci	Web(Continuous Integration)	网站
web-ci-59	Web	网站
web-ci-for-bid	Web	网站 (定制版)

定目录下。测试服务器上会有定时任务定时调用 Shell 脚本，Shell 脚本会监测应用文件的变化，或者版本定义文件中版本号的变化，任意一项有变动，则触发应用更新。版本变化通过读取版本配置文件来侦测，文件变化通过对比文件的 Hash 来判断。

Table 1.2: 自动部署服务器信息

IP	名称	备注
192.168.1.24	Jenkins 服务器	所有项目的构建、编译、打包
192.168.1.11	Nginx 服务器	文件的接收转发服务
192.168.1.6	App 测试服务器	所有 App 运行在此服务器

构建服务 (Auto Build)

构建服务包含获取源码更新、构建、构建后操作 3 步。由于项目针对不同应用场景，定义了不同的分支，所以在构建时，需要了解每个分支的含义。各分支定义如表1.3所示：

Table 1.3: 分支名称及含义

项目名称	分支名称	备注
system	v1.3	主分支,1 表示第几期，3 表示阶段
system	v1.3_api	接口分支
system	v1.3_exchange	数据交换分支
report	develop	主分支
report-frontend	develop	主分支

构建服务定时检查源码仓库的变动，监测到变动后，触发自动编译打包等操作。构建后，通过调用构建后脚本，将文件拷贝到服务器。构建后脚本路径/home/jenkins-bak/，各个脚本的含义如表1.4日所示。

脚本命名含义一般是项目名称加需要产生的作用，project-name 表示对应的项目名字，新增时可替换为项目的实际名称。after-build-handler 表示此脚本主要告诉 Jenkins 构建后需要执行此脚本完成一系列动作。构建需要特殊处理的地方：



Table 1.4: 自动部署后触发脚本命名规则

脚本名称	备注
{project-name}-after-build-handler.sh	项目后端构建后处理
{project-name}-frontend-after-build-handler.sh	项目后端构建后处理

```
1 #!/usr/bin/env bash
2
3 # 当使用未初始化的变量时，程序自动退出
4 set -u
5
6 # 当任何一行命令执行失败时，自动退出脚本
7 set -e
8
9 # 在运行结果之前，先输出执行的那一行命令
10 set -x
11
12 readonly APP_ID=""
13 readonly APP_KEY=""
14 readonly PROJECT_DIR="/var/lib/jenkins/workspace/web-ci-59"
15 readonly BUILD_OUTPUT_DIR="/var/lib/jenkins/workspace/web-ci-59/
    cms/target"
16
17 BUILD_DIST_FILENAME=${BUILD_OUTPUT_DIR}/cms.war
18
19 cd ${BUILD_OUTPUT_DIR}
20
21 #
22 # 文件过大，无法拷贝
23 # 将大文件拆分
24 #
25 split -b 80M ${BUILD_DIST_FILENAME}
```

```
26
27 CURRENT_TIME='date +%Y-%m-%d %H:%M:%S'
28
29 CONST_STR="ysf"
30
31 #
32 # 生成 TOKEN
33 # 接口各项认证参数的排列顺序是:
34 # 时间戳 (timestamp)、AppKey、随机字符串 (echostr)
35 #
36 TOKEN='echo -n ${APP_ID}${APP_KEY}${CURRENT_TIME}${CONST_STR}|
      md5sum|awk '{print $1}''
37 SEQUENCE_TOKEN='echo -n ${CURRENT_TIME}${APP_ID}${APP_KEY}${
      CONST_STR}|shasum -a 1|awk '{print $1}''
38
39
40 # 请求服务端，上传主文件
41 COMMAND='curl -H "APPID:$APP_ID" \
42 -H "TIMESTAMP:$CURRENT_TIME" \
43 -H "ECHOSTR:$CONST_STR" \
44 -H "TOKEN:$SEQUENCE_TOKEN" \
45 -F "file=@${BUILD_OUTPUT_DIR}/xaa" \
46 http://192.168.1.11:8083/api/fileExchange/upload'
47
48 COMMAND='curl -H "APPID:$APP_ID" \
49 -H "TIMESTAMP:$CURRENT_TIME" \
50 -H "ECHOSTR:$CONST_STR" \
51 -H "TOKEN:$SEQUENCE_TOKEN" \
52 -F "file=@${BUILD_OUTPUT_DIR}/xab" \
53 http://192.168.1.11:8083/api/fileExchange/upload'
```

在上传文件时，由于限制了文件大小 (估计是 100MB)，当部署文件大于 100MB 时，使用 split 命令，拆分成多个小文件分开上传，上传后通过 cat 命令组装成 war 包，

组装后的文件与原始文件一致，可以通过对比文件的 MD5 值来判断。

中转服务 (Forward Service)

中转服务将构建服务器上生成的应用包转发到应用服务器，应用包在测试服务器 192.168.1.6 的存放路径是 /home/deploy/credit。中转服务的配置主要是 Nginx 转发上传的请求到指定服务器：

```
1 location /api/fileExchange {
2     proxy_pass http:ip:port;
3     proxy_redirect off;
4 }
```

应用更新服务 (Update Trigger)

应用更新服务在测试服务器，通过建立 cron 定时任务，轮询检查应用更新情况。定时任务配置文件是 /etc/crontab，各个系统更新触发脚本路径是 /opt/app/script。定时任务配置示例：

```
1 #
2 # project name update trigger
3 #
4 */1 * * * * root /opt/app/script/project-name-trigger.sh
```

触发更新规则在相应的 Shell 脚本中实现。如下代码片段所示：

```
1 #!/usr/bin/env bash
2
3 # 部署触发器
4 # 定时检查文件修改
5 # 文件修改后，触发部署动作
6
7 # 当使用未初始化的变量时，程序自动退出
8 #set -u
9
10 # 当任何一行命令执行失败时，自动退出脚本
```

```
11 set -e
12
13 # 在运行结果之前，先输出执行的那一行命令
14 set -x
15
16 # 定义错误日志级别
17 LOG_LEVEL=-9000
18
19 #定义日志存放目录
20 SIMPLE_LOG_4_SH_DIR=/tmp/simplelog4sh
21
22 #导入日志
23 . /opt/app/script/log4shell.sh
24
25 # 当前运行程序版本
26 readonly APP_PATH="/opt/app/backend-v1.3"
27 readonly DEPLOY_PATH="/home/deploy/credit"
28 source ${APP_PATH}/report-version.properties
29 CURRENT_VERSION=${VERSION}
30 source ${DEPLOY_PATH}/report-version.properties
31 DEPLOY_VERSION=${VERSION}
32 readonly FILE_NAME="report-web-boot-${CURRENT_VERSION}.jar"
33 readonly DEPLOY_FILE_NAME="report-web-boot-${DEPLOY_VERSION}.jar"
34
35 logInfo "检查后端App更新..."
36
37 deploy()
38 {
39     logInfo "停止旧版本程序...,版本: ${CURRENT_VERSION}"
40     ps -ef|grep -w ${FILE_NAME}|grep -v grep|cut -c 9-15|xargs
41         kill 9
42     logDebug "开始拷贝新版程序文件....."
43     yes|cp -rf ${DEPLOY_PATH}/${DEPLOY_FILE_NAME} ${APP_PATH}
```



```
43     yes|cp -rf ${DEPLOY_PATH}/credit-report-version.properties ${  
        APP_PATH}  
44     logInfo "启动新版程序.....,程序路径:${APP_PATH},版本: ${  
        DEPLOY_VERSION}"  
45     ${APP_PATH}/start-v1.3.sh  
46 }  
47  
48 if test ${CURRENT_VERSION} = ${DEPLOY_VERSION}  
49 then  
50     logInfo "版本号无变化, 检查文件Hash...."  
51     CURRENT_VERSION_MD5='md5sum ${APP_PATH}/${FILE_NAME}|cut -d '  
        ' -f1'  
52     DEPLOY_VERSION_MD5='md5sum ${DEPLOY_PATH}/${FILE_NAME}|cut -d '  
        ' ' -f1'  
53     if test ${CURRENT_VERSION_MD5} != ${DEPLOY_VERSION_MD5}  
54     then  
55         deploy ""  
56     else  
57         logInfo "Hash无变化, 文件未修改, 后端App结束..."  
58     fi  
59 else  
60     logInfo "后端App版本有变化,开始部署新版程序..."  
61     deploy ""  
62 fi
```

由于目前应用较多, 单独列出每个应用的目录比较冗长。此处仅仅描述目录的命名的一般性原则。测试环境的目录统一在 `opt` 下 (项目初始阶段习惯), 正式环境的目录在根目录 `home` (初始习惯) 或者 `data` 下, `data` 目录存放应用是推荐的标准做法, 以后新应用部署推荐用此标准, `data` 下存放组织名称对应的目录, 组织名称下对应此组织相应的应用。当前各个应用的目录如表1.6所示。recommand 表示推荐做法, obsolete 表示过时的做法, 以后不推荐采用的方式。Test 表示测试环境, 对应的测试服务器 IP, Production 表示生产环境, 对应的生产环境的 IP。

Table 1.5: 项目部署目录信息

环境	目录	备注
Test	/opt/app/	部署目录
Test	/home/deploy/credit/	CI 部署文件存放目录
Test	/opt/app/script/	CI 部署脚本存放目录
Prod	/home/app/	obsolete
Prod	/data/{companyname}/app/	recommand

1.1.7 Nginx 超时转发

在后端有双机或多台机器提供服务的情况下，如果某一台机器超过了一定时间未响应，Nginx 将尝试将请求转发到下一台服务器。由于写操作时，如果没有在后端作重复校验，一旦写操作比较耗时，转发后会出现重复写的情况 (POST 一般不进行超时转发，但是幂等请求下也可能会产生重复，比如日志记录、触发消息通知等等)。避免此问题的一种设计方案是在前端渲染表单时，生成一个唯一 ID，防止重复提交。所以此处仅仅将超时转发的规则应用在部分 url 上，在 Nginx 中的配置如下：

```
1 upstream example_upstream{
2     server 192.168.0.1 max_fails=1 fail_timeout=3s;
3     server 192.168.0.2 max_fails=1 fail_timeout=3s backup;
4 }
5 location /example/ {
6     proxy_pass http://example_upstream/;
7     proxy_set_header Host: test.example.com;
8     proxy_set_head X-real-ip $remote_addr;
9     proxy_next_upstream error timeout http_500;
10 }
```

Nginx 在 POST, LOCK, PATCH 这种会对服务器造成不幂等的方法，默认是不进行重试的，如果一定要进行重试，则要加上如下配置：

```
1 # 非幂等也进行超时转发配置
```

```
2 proxy_next_upstream error timeout http_500 non_idemponent;
```

1.1.8 无法获取数据库连接

导致问题的原因：数据库连接配置过小，访问量增长时造成连接分配不足，导致无法顺畅访问站点。

最近 2 天网站运行一段时间后会突然宕掉，日志输出无法获取数据库连接 (Could not get JDBC connection)，初步推测是数据库连接泄漏，造成数据库可用连接被使用完，主备节点都拿不到连接，造成此问题，目前连接由连接池进行管理，一般情况下是不需要人工干预数据库的连接申请和释放，也有可能是某处进行了手工申请数据库连接，但是没有释放连接导致。使用如下 Shell 脚本每隔一段时间检测连接池数量：

```
1 # 每 5 秒查看一次到数据库连接个数
2 nohup watch -n 5 'lsof -i:5236|wc -l >> pool.log'
```

经过观察，数据库连接稳定在 300-310 之间，如果是连接池泄漏，那么连接会随着程序运行逐渐增长，最终达到连接数量上限，根据观察，初步可以排除连接泄漏导致此问题。后来经过日志筛查，发现如下输出：

```
1 # 活动连接数
2 active 20,maxActive 20
```

表示当前活动连接数 20 个，最大活动连接数 20 个。虽然到数据库的会话有 300 多个，但是绝大多数是空闲的 (Idle)，如下语句查看数据库当前会话的个数。

```
1 -- 查看数据库当前会话情况
2 select clnt_ip,state,user_name,count(*)
3 from v$sessions
4 group by clnt_ip,state,user_name
```

结果如图3.6所示：

可以看出，活动的会话稳定在 13 个，离最大个数 20 已经不远，极有可能是数据库会话配置过小，导致用户增多时，无可用连接分配。为了验证推断，使用 JMeter³模

³<http://jmeter.apache.org/>

拟多用户同时访问测试网站。逐步增加同时访问的用户的个数，当增加到 80 个时，网站无响应。检查后端日志，出现了线上的问题同样的日志输出，说明问题在此处，需要优化。DBCP 当连接数超过最大连接数时，所有连接都会被断开 [未找到权威出处]。经过考虑，逐步应用以下优化的方式。

增加网站数据库连接数量上限

调整配置增大数据库连接上限，可以增加网站并发服务能力：

```
1 <!-- 最小空闲连接数 -->
2 <property name="minIdle" value="50" />
3 <!-- 最大连接数 -->
4 <property name="maxActive" value="100" />
```

做了以上优化后，可以提高网站的并发服务量，降低由于访问量过大造成网站不可用的概率，20 个连接确实太小，综合评估数据库现有的连接资源 (500)，调整后连接数分配情况：

Table 1.6: 应用连接个数分配情况

环境	IP	连接个数
主机	192.168.1.1	100
备机	192.168.1.23	100
移动端	192.168.1.1	100
旧版 App、定制版 App、登陆、临时使用预留	192.168.1.*	200

增加了连接数上限后，为了验证效果，采用 crontab 定时任务每隔 5 分钟定时采集不同时间点数据库的活动 Session 个数：

```
1 # 每 5 分钟定时执行采集任务
2 */5 * * * * root /usr/local/dolphin/script/db-pool-monitor.sh
```

脚本将采集到的数据放入 csv 文件中，后期采用 Excel 生成可直观的图表：

```
1 #!/usr/bin/env bash
2
3 # 当使用未初始化的变量时，程序自动退出
4 set -u
5
6 # 当任何一行命令执行失败时，自动退出脚本
7 set -e
8
9 # 在运行结果之前，先输出执行的那一行命令
10 set -x
11
12 connect_size='lsof -i|grep 10.20.1.17|grep java|wc -l'
13 current_time='date "+%H:%M:%S"'
14 current_date='date "+%Y%m%d"'
15 echo "${current_time},${connect_size}" >> /home/monitor/dm-pool-"${current_date}".csv
```

数据库实时 Session 监控脚本向 csv 中写入 2 列，当前的时间和当前时间点的活动 Session 的个数，每天生成一个 csv 文件。根据 csv 生成的折线图效果如图3.7所示。

8:50 到 9:40 时间段进行了压力测试，这个时间范围内数据库的活动 Session 有明显的增加。

增大数据库 Session 数量上限

目前数据库的会话数量是 500，可以调整为 1000。

缓存页面信息

实际上，页面在绝大部分时间是不会改变的，完全可以将页面信息缓存在 Redis 中，不但减轻数据库压力，还可成倍提高网站响应速度和并发处理能力。此种优化方式需要对代码做一定程度改造，比如发布信息时，需要清除对应模块的 Redis 缓存信息，保证信息更新及时。

将流量分配到不同的机器

后端流量可以主备机同时启用，涉及到 Session 共享问题，需要进一步测试。

1.1.9 Java heap space

导致问题的原因：表面看是 APP 分配的内存资源不足，系统并发性能不足，应对持续并发请求能力需要加强，实际是 Tomcat 请求头配置过大 (100MB)，每个线程会申请对应大小的空间，而大对象 JVM 直接会放到老年代，一旦并发请求稍多，造成内存溢出。

上周网站 Down 掉，本周接口 Down 掉，愉快的周末泡汤了。所以，暗自忖度，一定要找到问题所在。Heap 的大小是 Young Generation 和 Tenured Generation 之和。在 JVM 中如果 98% 的时间是用于 GC，且可用的 Heap size 不足 2% 的时候将抛出此异常信息。内存溢出一般分为 2 种情况，一时是超出预期的访问量/数据量，另一种是内存泄露 (Memory leak)。第一次的内存溢出异常有 hprof 文件，将文件拷贝出来进行分析。第二次接口异常没有生成 hprof，采用 VisualVM⁴分析内存情况。新建

```
1 grant codebase "file:/home/local/jdk1.8.0_111/lib/tools.jar" {  
2     permission java.security.AllPermission;  
3 };
```

在服务端启动 jstatd (Java State Daemon) 守护进程：

```
1 # 启动统计守护进程  
2 ./jstatd -J-Djava.security.policy=../jstatd.all.policy -J-Djava.  
    rmi.server.hostname=10.10.1.53  
3 # 生成内存 dump 文件  
4 jmap -dump:format=b,file=/opt/example.dump [pid]
```

在本地使用 VisualVM 监视服务端 JVM 的运行情况，内存的增长速度在 4-5MB/s，2GB 的内存 4-5min 就被消耗完毕，接着进行一次 YGC (Young Garbage Collection)。初步监测，老年代 (Old Generation) 的内存的确在不断增长。但是仅凭此点，无法下结论一定存在内存泄漏，如果老年代在以后执行 GC (Garbage Collection) 后，老年代 (Old Generation) 初始内存不停增长，才可以推断的确存在内存泄漏，因为不停增加的初始

⁴VisualVM 在 JDK 安装包中附带，在 \$JAVA_HOME/bin 目录下。

内存，标志着有部分内存永远无法被 FGC 回收，随着程序的不断运行，必定会由于内存不足而 Hang 住。

Table 1.7: 应用内存消耗

并发数量	Old Generation 内存消耗	连接个数
0	404MB	100
1	1.119GB	100
10	3.847GB	100
20	3.689GB	100
30	4.834GB	100

在一段时期内，程序一启动老年代就已经满了，占用率直接 100%。开始是以为流量较高，YGC 频繁，不停的有经过多次 YGC 未回收的对象移入老年代，最多 FGC 执行频繁一些罢了。后来更为诡异的事情发生了，机器突然变得非常卡顿，磁盘使用量快速增长，最后程序直接无法启动，启动不到 30s 立即停止响应或者直接输出 Java Heap space 错误。

问题复现

经过备份的日志阅读，发现一段时间段内，有大量针对某一查询请求，且非常集中。下午抽出时间在测试环境使用 JMeter 构造红集中请求的查询，模拟线上场景。当并发的用户量不断增长时，YGC 越来越频繁，同时老年代初始内存消耗上升快速。当并发量达到 50 时，老年代内存使用量飙升到 100%，同时程序 Hang 住，输出 Java Heap space 错误。终于知道症结在机器在应对并发流量时内存不足，没有对内存做好估算。并发量升高时，应用对的内存消耗极大的超出了预期，即使每个线程 5MB 的内存，100 个并发也才 500MB，当前的内存有 5GB，按照预期是完全足够的。但最终还是将服务器的内存增大到 64GB 来解决问题。

问题回顾

当在并发量较高时，APP 启动时，会占用比较多的老年代内存，所以在下午应用已启动老年代内存立即耗尽，年轻代不停的执行 GC。当并发超出系统承载能力时，老年代默认内存已经无法支撑应用启动的初始内存消耗，所以应用启动立即发生内存溢出。有于在发生内存溢出时会生成 dump 文件，将现场保留到磁盘以备问题排查，每

个 dump 文件从 2GB-8GB 不等，所以造成磁盘使用量急速消耗。

```
1 # 内存溢出时生成 dump 文件
2 -XX:+HeapDumpOnOutOfMemoryError
3 -XX:HeapDumpPath=/home/app
```

不停的尝试启动，不停的生成 dump 文件，造成磁盘 IO 急剧升高，所以机器卡顿。

问题解决

一是增大系统并发吞吐量，二是系统需要有限流和熔断机制。为了避免类似情况，系统做如下优化：

- 一是将热点查询做成静态化页面，针对一级页面，打开即默认加载的数据静态化，避免后端接口处理，降低接口压力。
- 二是缓存凭据，凭据信息更改不频繁，直接缓存到内存中，避免数据库查询。
- 三是评估内存消耗，针对设计预期的并发分配相应的资源。
- 四是限流，识别无效请求并主动丢弃，减少无效流量对资源的占用和消耗，可以根据实时流量调整限制级别。例如在平时网站运行时，可以将限制关闭，在应对重大事件时，可以将限制开启并根据网站保障级别调整限制级别。
- 五是使用监控工具，对网站的实时流量和服务器负载有监控预警，根据流量情况采取不同的措施。

关键的问题是，在当前并发的情况下，不应该消耗如此大的内存。问题在哪里？使用 JMap⁵(Java Memory Map) 命令将运行时内存情况输出，采用 MAT⁶(The Eclipse Memory Analyzer) 工具进行分析，查看内存中的大对象如图3.8所示。

从 Histogram 图中可以看出，对象中 byte 数组的大小超过了 8GB，每一个 byte 数组元素大小是 102408208(约 102MB)。怎么会有如此大对象？问题非常有可能就出在这些大对象上面了。下载 Tomcat 8.0.53(注意下载的源码需要尽量服务器上运行的版本一致，InternalNioInputBuffer 类在往后版本的 Tomcat 已经被移除) 源码查看，找到如下代码片段：

```
1 @Override
2 protected void init(SocketWrapper<NioChannel> socketWrapper,
3 AbstractEndpoint<NioChannel> endpoint) throws IOException {
```

⁵<https://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html>

⁶<https://www.eclipse.org/mat/>


```
4
5     socket = socketWrapper.getSocket();
6     if (socket == null) {
7         // Socket has been closed
8         // in another thread
9         throw new IOException(sm.getString("iib.socketClosed"));
10    }
11    socketReadBufferSize =
12    socket.getBufHandler().getReadBuffer().capacity();
13
14    int bufLength = headerBufferSize + socketReadBufferSize;
15    if (buf == null || buf.length < bufLength) {
16        buf = new byte[bufLength];
17    }
18
19    pool = ((NioEndpoint)endpoint).getSelectorPool();
20 }
```

在处理请求的线程初始化时,根据请求头大小新构建一个 byte 数组 (`new byte[bufLength]`),原来是请求头设置过大。系统在生成 byte 数组时,判断是大对象,直接将对象放到老年代中,造成老年代消耗内存急速增长。

1.1.10 报表在现场演示环境无法打开

导致问题的原因：报表地址配置的内网地址，公网用户无法访问。

报表服务部署在内网的机器上，内网的机器与平时使用的机器在一个局域网中，内网用户访问报表可以直接通过内网 IP。但是公网的用户无法访问报表服务器的内网 IP，需要通过公网 IP 做一次转发。就像在自己家里部署一个服务，但是在办公室的 PC 是无法访问的，需要有公网 IP 做转发。公网用户访问报表的一般流程是，先访问公网地址，公网地址将流量转发到内网的反向代理服务器，反向代理服务器根据报表的 URL 将流量路由到报表服务器。使用如下语句更新报表配置即可：

```
1 # 将报表服务的IP调整为公网IP
2 update report_config
```

```
3 set url = replace(url,'内网IP','公网IP')
```

1.1.11 查询变慢

导致问题的原因：日志数据不断膨胀造成查询变慢，日志记录的方式需要优化。

这两天不断有反馈原来很快的查询现在不知道什么原因变慢了，查看了变慢的页面后，确实速度不符合设计预期，正常情况下不应大于 200ms，平均响应时间肯定不会超过 1s，页面数据是放在接口缓存中。加上调用链路的开销，不应该有延迟。访问接口数据的流程如图3.10所示：

首先确定接口缓存有效，在接口服务器上请求接口服务：

```
1 curl -H "APPID:" \  
2 -H "TIMESTAMP:2016-12-19 16 (tel:2016121916):58:02" \  
3 -H "ECHOSTR:sdsaasf" -H "TOKEN:" \  
4 -H "Accept: application/json, text/plain" \  
5 http://192.168.1.11:28081/api/example?name=\&page=1\&size=10|jq  
  , ,
```

在 50ms 内可以返回结果，说明缓存有效，不是从数据库获取。其次确定网络设备影响(已经感受到的发送 POST 请求会有 1-2s 左右延迟，网络 A 的服务器发出请求，网络 B 的服务器需要 1-2s 才能实际接收到并处理)，在网络 A 使用访问接口服务，时间与网络 B 访问相近。由于端口 80 绑定域名，新建内网服务，从内网站点访问接口，时间超过 3s，说明问题在 A 网的站点。后经过跟踪，发现 A 网站点接口本身处理非常快，但是绝大部分时间消耗在处理请求后返回的过程中。经过跟踪，处理请求后查询更新了日志表，而日志表的数据在 800W 左右。

问题解决

- 快速的解决方式，临时备份日志，建立空的日志表。
- 重构日志，直接记录流水，不记录统计信息。(Recommend)
- 构建独立的日志服务，所有的日志打到独立的服务上，采用简单消息队列。(Recommend)
- 尝试 ELK 日志分析系统。

1.1.12 资源分配约定

由于最近处理的问题大多涉及到资源问题，例如文件无法上传，邮件无法发送，主要由于磁盘空间不足。原来分配的虚拟机一般 home 目录较大，所以将随着程序运行可能需要消耗较大磁盘空间的目录放在 home 目录下，但是新内网虚拟机恰好 home 目录很小。导致一些列问题，所以这里约定新部署 App 目录的统一规范。

Table 1.8: 磁盘目录分配约定

路径	用途
/opt/alibaba/app	存放项目应用程序
/opt/alibaba/var/data	存放数据文件
/opt/alibaba/var/image	存放图片文件
/opt/alibaba/local	存放三方程序
/opt/alibaba/backup	存放备份文件
/opt/alibaba/logs	存放日志文件

考虑到安全，程序启动须使用 alibaba 帐号，不使用 root 启动应用程序。日志应独立存放，避免放到应用目录下，因为日志文件通常会比较大，独立存放日志好处之一是在新应用部署时，可以直接拷贝 app 目录，而避免与日志一起拷贝或者需要单独做一步忽略日志的操作。

1.1.13 Tomcat 请求头设置

App 在启动的时候占用老年代内存较大，当并发增大时，消耗内存迅速增加。经过排查，是 maxHttpHeaderSize 参数配置过大。

```
1 #
2 # 请求头允许的最大长度，单位为 KB
3 # 如果没有指定，默认是 8192(8KB)
4 # 这里请适当斟酌，在批量查询时
5 # URI 中需要带参数（批量查询的参数会比较大）
6 # 点击返回按钮后
7 # 需要根据 URI 中记录的参数定位到上一个页面
8 # 如果不增加 Header Size
```

```
9 # 没有更好的方案实现返回功能
10 # 所以 Header Size 设置得较大
11 #
12 server.max-http-header-size=102400000
```

这种问题应该是很明显的，为何到现在才发现？对比了旧版的配置，旧版配置的是 1MB。估计在新版配置的时候顺手将开发环境的配置拷贝到生产，而开发环境的配置比较随意。Tomcat 在处理 HTTP 请求时，会直接申请 `maxHttpHeaderSize` 大小的内存空间，而不仅仅是一个限制。

1.1.14 GoAccess

最近网站流量“偏高”，导致关键服务停服。所以可以考虑实时的监控网站的健康状态，作出对应的决策。由于安全原因不能使用监控应用，所以找到一个直接在终端下可以查看 Nginx 日志统计信息的应用 GoAccess⁷。

```
1 goaccess -a -d -f /usr/local/nginx/logs/example.log
2 goaccess -a -d -f /usr/local/nginx/logs/example.log -p /etc/
  goaccess.conf -o /data/html/hexo/public/go-access.html
```

不能以服务的方式运行，所以做一个 `crontab` 定时任务，每隔一段时间生成网站访问日志的统计信息：

```
1 # 每 20 分钟执行
2 */20 * * * * root goaccess -d -f /usr/local/nginx/logs/example.
  log -p /usr/local/etc/goaccess.conf
```

根据 URL 的访问频率统计信息如图 3.14 所示，从统计图中可以看出，九成以上的请求都集中在一个 URL。

根据 IP 的统计信息如图 3.15 所示，从统计图中可以看出，5 成以上的访问请求只服务了 3 个 IP。

⁷<https://goaccess.io>

1.1.15 Nginx 启用压缩

最近演示时说系统很慢，打开系统查看，首屏加载时间超过 8s，确实不符合常规。由于网站是 SPA 应用，首先查看 Javascript 大小，竟然有接近 8MB 大小，对比了线上系统，发现演示系统 Javascript 没有压缩。这里经过了二级 Nginx 代理，内网访问 Nginx 代理，浏览器中查看 Javascript 大小在 1.6MB 左右，说明内网的 Nginx 代理已经压缩了 Javascript 文件，推测可能是另一级需要对代理 Nginx 做启用 gzip 压缩配置，尝试在另一级 Nginx 代理添加如下配置：

```
1 # HTTP 代理版本 (proxy_http_version)
2 proxy_http_version 1.1
3 # gzip 支持版本 (gzip_http_version)
4 gzip_http_version 1.0
```

gzip 模块默认支持压缩的 HTTP 版本是 1.1，而代理模块的默认代理 HTTP 版本是 1.0⁸，不匹配，造成 gzip 的压缩没有生效，调整任意一项即可。

1.1.16 查看网站并发

使用如下命令查看网站并发：

```
1 netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}'
2
3 # 查看 Ngni 并发
4 tail -f access.log | awk -F '[' '{print $2}' | awk 'BEGIN{key="";
    count=0}{if(key==$1){count++}else{printf("%s\t%d\r\n", key
    , count);count=1;key=$1}}'
```

结果如图3.13所示：

```
1 goaccess -a -d -f /data/logs/fanhaobai.com.access.log -p /etc/
    goaccess.conf -o /data/html/hexo/public/go-access.html
```

⁸<https://reinout.vanrees.org/weblog/2015/11/19/nginx-proxy-gzip.html>

不能以服务的方式运行，所以做一个 crontab 定时任务，每隔一段时间生成网站访问日志的统计信息：

```
1 # 每 20 分钟执行
2 */20 * * * * root goaccess -d -f /usr/local/nginx/logs/example.
    log -p /usr/local/etc/goaccess.conf
```

根据 URL 的访问频率统计信息如图3.14所示，从统计图中可以看出，九成以上的请求都集中在一个 URL。

根据 IP 的统计信息如图3.15所示，从统计图中可以看出，5 成以上的访问请求只服务了 3 个 IP。

1.1.17 自动构建更新的问题

有时经常会遇到代码提交后在测试环境总看不到效果的问题。可以使用如下方式逐步排查。第一步检查 Jenkins 是否获取到了最新的代码，在 Jenkins 构建目录 (/var/lib/jenkins/workspace) 下检查最新代码片段。第二步检查 Tomcat 目录下是否包含最新的调整后的代码，由于 Tomcat 下是已经编译完毕的 class 文件，需要借助 Luyten⁹工具打开，查看源码，是否包含最新的改动，如果包含最新改动，说明自动编译构建没问题。这里查看 Tomcat 下的 class 文件包含了最新的调整，但是没有自动更新，重启 Tomcat 后更新生效。尝试调整发布脚本，每次发布时重新启动 Tomcat 来解决此问题。

```
1 #!/usr/bin/env bash
2
3 # 当使用未初始化的变量时，程序自动退出
4 set -u
5
6 # 当任何一行命令执行失败时，自动退出脚本
7 set -e
8
9 # 在运行结果之前，先输出执行的那一行命令
10 set -x
11
```

⁹Luyten 是一个出色的 Java 反编译器，项目地址是<https://github.com/deathmarine/Luyten>。

```
12 readonly AUTO_BUILD_OUTPUT_PATH="/var/lib/jenkins/workspace/  
    dolphin/api/target"  
13 readonly PROGRAM_NAME="cms.war"  
14 readonly DEPOLY_PATH="/opt/alibaba/local/apache-tomcat-8.0.52/  
    webapps/"  
15 readonly TOMCAT_BIN_PATH="/opt/alibaba/local/apache-tomcat-8.0.52/  
    bin"  
16 readonly APP_FILE_IDENTITY_NAME="apache-tomcat-8.0.52"  
17  
18 # 停止站点  
19 ps -ef|grep -w ${APP_FILE_IDENTITY_NAME}|grep -v grep|cut -c 9-15|  
    xargs kill 9  
20  
21 sleep 3  
22  
23 # 拷贝部署文件  
24 yes|cp -rf ${AUTO_BUILD_OUTPUT_PATH}/cms.war ${DEPOLY_PATH}  
25  
26 # 启动站点  
27 ${TOMCAT_BIN_PATH}/startup.sh
```

1.1.18 Non-resolvable parent POM for Could not find artifact

Maven 构建时提示如下错误：

```
1 Non-resolvable parent POM for Could not find artifact and 'parent.  
    relativePath' points at wrong local POM
```

在 Windows->Proference->Maven->User Settings 中，调整 Maven 的配置文件的位
置为 D:/maven3/maven3/conf/settings.xml，重新构建即可。另外在项目中，某一个项目
中没有 Maven Dependency，导致依赖包无法找到，在项目的.classpath 文件中添加如下
配置即可：

```
1 <classpathentry kind="con" path="org.eclipse.m2e.  
    MAVEN2_CLASSPATH_CONTAINER">  
2     <attributes>  
3         <attribute name="maven.pomderived" value="true"/>  
4         <attribute name="org.eclipse.jst.component.nondependency"  
            value=""/>  
5     </attributes>  
6 </classpathentry>
```

.classpath 文件用于记录项目编译环境的所有信息,包括:源文件路径、编译后 class 文件存放路径、依赖的 jar 包路径、运行的容器信息、依赖的外部 project 等信息。另查看 Maven Repositor 在菜单 Window->Show View->Other...->Maven->Maven Repositories 下。

1.1.19 LocalDateTime 格式化

在 Spring 中,接收 LocalDateTime 日期时间数据时,只需要使用 @DateTimeFormat 注解即可。在没有引用 jsr310(Java Specification Requests) 包时:

```
1 compile ("com.fasterxml.jackson.datatype:jackson-datatype-jsr310:2  
    .5.1")
```

返回的日期序列化后如下:

```
1 {  
2     "message": "ok",  
3     "code": 2000,  
4     "data": {  
5         "dtexample": {  
6             "hour": 23,  
7             "minute": 11,  
8             "second": 16,  
9             "nano": 350000000,  
10            "dayOfYear": 255,  
11            "dayOfWeek": "WEDNESDAY",
```



```
12         "month": "SEPTEMBER",
13         "dayOfMonth": 12,
14         "year": 2018,
15         "monthValue": 9,
16         "chronology": {
17             "calendarType": "iso8601",
18             "id": "ISO"
19         }
20     }
21 }
22 }
```

jsr310 提供了对 `java.util.Date` 的替代，另外还提供了新的 `DateTimeFormatter` 用于对格式化/解析的支持。JSR 310 规范领导者 Stephen Colebourne 就是 `joda-time` 作者，其主要思想也是借鉴了 `joda-time`，而不是直接把 `joda-time` 移植到 Java 平台中。添加 `jsr` 包没有加注解的情况下，返回的时间格式如下：

```
1 {
2     "message": "ok",
3     "code": 2000,
4     "data": {
5         "dtexample": [
6             2018,
7             9,
8             12,
9             22,
10            12,
11            45,
12            347000000
13        ]
14    }
15 }
```

在 Model 的字段上添加时间格式的注解：

```
1 @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd  
   HH:mm:ss", timezone = "GMT+8")
```

返回的时间格式如下¹⁰：

```
1 {  
2   "message": "ok",  
3   "code": 2000,  
4   "data": {  
5     "dtexample": "2018-09-12 22:13:16"  
6   }  
7 }
```

1.1.20 invalid constant type: 18

Tomcat 启动项目时提示错误：invalid constant type: 18，原因是 Java 8 下运行项目需要 javassist 3.18.2-GA 及以上版本¹¹，在 hibernate entity manager 包中的 AbstractJarVisitor 类中有调用，添加依赖：

```
1 <dependency>  
2   <groupId>org.javassist</groupId>  
3   <artifactId>javassist</artifactId>  
4   <version>3.18.2-GA</version>  
5 </dependency>
```

CGLib 的底层基于 ASM 实现，是一个高效高性能的生成库；而 ASM 是一个轻量级的类库，但需要涉及到 JVM 的操作和指令；相比而言，Javassist 要简单的多，完全是基于 Java 的 API，但其性能相比前二者要差一些。

¹⁰ 参考文章：<https://geowarin.com/correctly-handle-jsr-310-java-8-dates-with-jackson/>

¹¹ <https://stackoverflow.com/questions/24281235/error-creating-entitymanagerfactory-due-to-error-tying-to-scan-jar-file>

1.1.21 视频解析

有时需要获取腾讯视频等地址，可以使用点量视频解析¹²，可以获取到视频到真实地址，腾讯视频在下载一段时间后，速度会降到几 KB 左右。

1.2 编辑中

1.2.1 缓存雪崩 (Cache stampede)

当系统在高负载时，缓存失效后，导致大量的查询请求打在后端数据库上，导致数据库不堪负载，导致宕机。避免缓存雪崩可以采用如下方式。

锁 (Lock)

获取缓存 key 时，加锁。加锁减轻了数据库的压力，但是并没有提高系统吞吐量。锁住的时候，其他的线程都在等待。

设置过期标志更新缓存

1、缓存标记：记录缓存数据是否过期，如果过期会触发通知另外的线程在后台去更新实际 key 的缓存；

2、缓存数据：它的过期时间比缓存标记的时间延长 1 倍，例：标记缓存时间 30 分钟，数据缓存设置为 60 分钟。这样，当缓存标记 key 过期后，实际缓存还能把旧数据返回给调用端，直到另外的线程在后台更新完成后，才会返回新缓存¹³。

每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存。

为 key 设置不同的缓存失效时间

1.2.2 TCPCopy

这里需要实现生产服务器抓取到流量，在测试服务器上进行回放，回放时可以适当放大流量，使程序实际具有更强的处理能力。TCPCopy 可分为在线模式和离线模式，离线模式首先在线上服务器抓包，再拷贝到测试服务器进行重放。使用命令 dump 流量：

```
tcpdump -i eth0 -w /tmp/xxx.cap
```

¹²<http://old.flvurl.cn/>

¹³<http://www.cnblogs.com/leeSmall/p/8594542.html>

1.2.3 Could not get a resource from the pool

网站运行一段时间后，无法访问，查看日志，出现如下提示：

```
1 redis.clients.jedis.exceptions.JedisConnectionException: Could not
   get a resource from the pool
2 ...
3 Caused by: java.util.NoSuchElementException: Pool exhausted
4 at org.apache.commons.pool2.impl.GenericObjectPool.borrowObject(
   GenericObjectPool.java:464)
```

JedisPool 中的 Jedis 对象个数是有限的，默认是 8 个。

<https://yq.aliyun.com/articles/236384>

1.2.4 服务限流

由于服务器的硬件资源和网络带宽资源始终是有限的，服务器可以承载的流量也会有上限，超过应用的承载极限后，程序稳定服务即无法保障。所以除了提高服务器服务能力外，还需要在超过服务器的承载上限之前，限制过量的访问请求，保护应用。网站许多内容缓存在 Redis 中，主要保障访问数据库连接不要超过数据库提供的最大连接上限（目前数据库提供的会话为 100），或者说不导致数据库过载。接口应用不要超过接口所能提供的最高并发。接口服务可以在 Nginx 通过 `ngx_http_limit_req_module` 配置来实现，保障打到接口的流量不会超过接口可以承载的最大流量（经初步测试，在内存 40GB 左右的情况下，目前接口并发能到 100 左右，具体数量要根据接口的业务员逻辑复杂情况而定）。同时网站侧需要作流量统计识别与限制，防止无效流量占用接口资源导致接口疲于应付爬虫等无效流量而无法服务于正常用户。

网站流量识别与限制

网站不采取在 Nginx 上做流量限制，原因是

接口 Nginx 防止流量过载

优化后，在测试环境作压力测试检验。

1.2.5 Python 绘图

<https://www.jianshu.com/p/d9cc124d8a30>


```
1 pip install numpy
2 pip install scipy
3 dnf install python-matplotlib
```

1.2.6 XX-Net

redis 为了避免客户端连接数过多，有一个 `timeout` 配置，意思是如果连接的空闲时间超过了 `timeout` 的值，则关闭连接。默认配置是 0，意思是没有超时限制，永远不关闭连接。

1.2.7 fluxion

Handshake Snooper 进行抓包嗅探。Monitor 哦被动监听模式，只有对方主动连接目标时才会，抓到包，比较被动，不确定性很大，但是没有攻击性。第二种是常用的方式，使用 Aircrack-ng 强制解除认证，迫使目标进行重新连接。第三种类似第二种方式。



2. DB

2.1 Redis

默认 Redis 会以快照的形式将数据持久化到磁盘的 (一个二进制文件, `dump.rdb`, 这个文件名字可以指定), 在配置文件中的格式是: `save N M` 表示在 `N` 秒之内, Redis 至少发生 `M` 次修改则 Redis 抓快照到磁盘。当然也可以手动执行 `save` 或者 `bgsave` (异步) 做快照。Redis 持久化分别有 RDB 和 AOF 两种方式, Append-Only 方法可以做到全部数据不丢失, 但 Redis 的性能就要差些。AOF 就可以做到全程持久化, 只需要在配置文件中开启 (默认是 `no`), `appendonly yes` 开启 AOF 之后, Redis 每执行一个修改数据的命令, 都会把它添加到 AOF 文件中, 当 Redis 重启时, 将会读取 AOF 文件进行“重放”以恢复到 Redis 关闭前的最后时刻。

2.1.1 Redis 消息队列

Redis 不是为消息队列而设计的, 其实不是做消息队列的理想选择, 不过 Redis 简单, 能够满足目前的需求, 姑且用之。Redis 的 PUB/SUB 机制, 即发布-订阅模式。利用的 Redis 的列表 (lists) 数据结构。比较好的使用模式是, 生产者 `lpush` 消息, 消费者

brpop 消息，并设定超时时间，可以减少 redis 的压力。只有在 Redis 宕机且数据没有持久化的情况下丢失数据，可以根据业务通过 AOF 和缩短持久化间隔来保证很高的可靠性，而且也可以通过多个 client 来提高消费速度。但相对于专业的消息队列来说，该方案消息的状态过于简单（没有状态），且没有 ack 机制，消息取出后消费失败依赖于 client 记录日志或者重新 push 到队列里面。RocketMQ 没有成熟稳定的 Python 客户端，有一个小众客户端但是不支持 Python3，RabbitMQ 貌似主要由商业公司主导，虽然成熟稳定，但是使用貌似不够广泛，Kafka 虽然主要的使用场景是大规模流式数据处理，但是考虑到使用的广泛性，和成熟的周边，还是选择 Kafka，以后的日志处理估计也会用到它。

2.2 SSDB

Postgresql 持久化时速度跟不上抓取的速度，需要一个消息队列，第一是写入队列中数据能够落地不能丢，第二是在写入性能不高的情况下，可以缓冲足够庞大的数据，毕竟无法时刻监控持久化到数据库的任务。而 SSDB 恰好可以满足需求，redis 讲数据全部存储在了内存中，而云服务器内存不够大，在队列数据较多时无法有良好的表现，而 SSDB¹ 绝对优势是将数据写入了硬盘，同时也支持配置部分内存用于缓存热数据。经过试用，发现文档太过于简洁，缺乏配套的工具，比如客户端就只有 PHP SSDB Admin，客户端部署困难。

2.3 Postgres

查看 Postgresql 日志：

```
1 tail -f /usr/local/var/postgres/server.log
```

在 Mac 上启动 Postgres：

```
1 # 命令启动 Postgresql
2 sudo brew services start postgresql
3 # /usr/local/var/postgres/dolphinpg 为数据目录
4 /usr/local/Cellar/postgresql/9.6.3/bin/pg_ctl -D /usr/local/var/
   postgres -l /usr/local/var/postgres/server.log start
```

¹http://ssdb.io/zh_cn/


```
5 /usr/local/Cellar/postgresql/10.5/bin/pg_ctl -D /usr/local/var/  
    postgres/dolphinpg -l /usr/local/var/postgres/server.log  
    start
```

原来的数据列是 integer 类型，调整列的数据类型为 varchar：

```
1 -- 将列类型调整为 varchar  
2 alter table book alter column douban_id type varchar(64);
```

fdw 是 foreign-data wrapper 的一个简称，可以叫外部封装数据，可以在本地数据库操作远程数据库。创建远程 server：

```
1 create server server_remote  
2 FOREIGN data wrapper postgres_fdw  
3 OPTIONS(host '127.0.0.1', port '5432', dbname 'dolphin');  
4  
5 -- 查看所有远程连接  
6 SELECT * from pg_foreign_server;
```

在 server_remote 下为角色 postgres 创建一个用户匹配信息，options 里是用户名和密码。

```
1 create user mapping for postgres  
2 server server_remote  
3 options(user 'postgres',password 'postgres');
```

现在远程连接中需要连接的数据库为 dolphin，需要用到的表为 book，因此在完成上述创建后，就可以在本地创建外部表，表中的记录则完全和远程表中一样。远程 book 表和本地 book 表的新增都会同步到对方。创建关联表：

```
1 CREATE FOREIGN TABLE fund_words_lib(word varchar,remark varchar,id  
    bigint)  
2 server server_remote  
3 options (schema_name 'public',table_name 'fund_words_lib');
```

本地的表和远程的表更新都会同步到对方。

2.3.1 CTID

ctid: 表示数据记录的物理行当信息, 指的是一条记录位于哪个数据块的哪个位
移上面。格式 (blockid,itemid): 拿其中 (0,1) 来说; 0 表示块 id; 1 表示在这块第一条
记录。Difference against Oracle is that CTID is unique only inside particular table. Because
PostgreSQL have separate databases on one server and stores every table in separate file on
the disk. Note that although the ctid can be used to locate the row version very quickly, a
row's ctid will change each time it is updated or moved by VACUUM (vacuum: 回收未显示
的物理位置; 标明可以继续使用。) FULL. Therefore ctid is useless as a long-term row
identifier. The OID, or even better a user-defined serial number, should be used to identify
logical rows.

2.3.2 常用 SQL

查看单个表大小:

```
1 -- 数据库中单个表的大小 (不包含索引)
2 select pg_size_pretty(pg_relation_size('douban_book_id'));
```

查看 Postgresql 的锁:

```
1 SELECT
2     procpid,
3     start,
4     now() - start AS lap,
5     current_query
6 FROM
7     (SELECT
8         backendid,
9         pg_stat_get_backend_pid(S.backendid) AS procpid,
10        pg_stat_get_backend_activity_start(S.backendid) AS start,
11        pg_stat_get_backend_activity(S.backendid) AS current_query
12     FROM
13         (SELECT pg_stat_get_backend_idset() AS backendid) AS S
```

```
14 ) AS S ,pg_stat_activity pa
15 WHERE
16     current_query <> '<IDLE>' and procpid<> pg_backend_pid() and
17     pa.pid=s.procpid and pa.state<>'idle'
18 ORDER BY
19     lap DESC;
```

查询出 PID 后，直接 kill 即可。为表新增 id 自增序列：

```
1 -- 为表新增自增序列
2 alter table public.spider_urls_pool alter column id set default
   nextval('public.scrapy_urls_id_seq');
```

新增唯一约束：

```
1 -- 爬取 URL 新增唯一约束
2 alter table spider_urls_pool add constraint
   uk_spider_urls_pool_unique_scrapy_url unique (scrapy_url);
```

去重

去除 fund_words_lib 表的重复数据²：

```
1 delete from fund_words_lib fwl
2 where fwl.ctid <>
3 (
4     select min(fwl_emb.ctid)
5     from fund_words_lib fwl_emb
6     where fwl.word = fwl_emb.word
7 )
```

此语句相当缓慢，6W 多数据用了将近一个小时，慢的可以。

²<https://stackoverflow.com/questions/6583916/delete-duplicate-records-in-postgresql>

2.3.3 WAL(Write-Ahead Logging)

PostgreSQL 在写入频繁的场景中，会产生大量的 WAL 日志，而且 WAL 日志量会远远超过实际更新的数据量。我们可以把这种现象起个名字，叫做“WAL 写放大”，造成 WAL 写放大的主要原因有 2 点。

1. 在 checkpoint 之后第一次修改页面，需要在 WAL 中输出整个 page，即全页写(full page writes)。全页写的目的是防止在意外宕机时出现的数据块部分写导致数据库无法恢复。

2. 更新记录时如果新记录位置(ctid)发生变更，索引记录也要相应变更，这个变更也要记入 WAL。更严重的是索引记录的变更又有可能导致索引页的全页写，进一步加剧了 WAL 写放大。

过量的 WAL 输出会对系统资源造成很大的消耗，因此需要进行适当的优化。

2.3.4 跨库查询

PostgreSQL 跨库访问有 3 种方法：Schema，dblink，postgres_fdw，跨库查询示例如下：

```
1 insert into fund_words_lib(word,remark,id)
2 select *
3 from dblink('host=127.0.0.1 dbname=dolphin user=postgres password=
   postgres',
4           'select * from fund_words_lib where id > 69999 and id
   < 80000')
5 as fund_words_lib(word varchar,remark varchar,id bigint)
```

dblink 执行一个远程查询命令。dblink is a module that supports connections to other PostgreSQL databases from within a database session.postgres_fdw 远程可写功能是 9.3 版本出来后才新加的，而 dblink 也可以在以前的 postgresql 版本中使用。

2.3.5 统计信息

pg_stat_statements 模块提供一种方法追踪一个服务器所执行的所有 SQL 语句的执行统计信息，可以用于统计数据库的资源开销，分析 TOP SQL。

```
1 -- 赋给执行函数权限
```

```
2 grant execute on function pg_stat_statements_reset() to postgres;
3
4 -- 放弃到目前为止搜集的所有统计
5 -- 默认需超级用户权限
6 select pg_stat_statements_reset();
```

2.3.6 异步流复制 (Async Stream Replica)

基于 Standby 的异步流复制，这是 PostgreSQL 9.x 版本（2010.9）之后提供的一个功能，类似的功能在 Oracle 中是 11g 之后才提供的 active dataguard 和 SQL Server 2012 版本之后才提供的日志传送。PostgreSQL 在数据目录下的 pg_xlog 子目录中维护了一个 WAL 日志文件，该文件用于记录数据库文件的每次改变，这种日志文件机制提供了一种数据库热备份的方案，即：在把数据库使用文件系统的方式备份出来的同时也把相应的 WAL 日志进行备份，即使备份出来的数据块不一致，也可以重放 WAL 日志把备份的内容推到一致状态。这也就是基于时间点的备份（Point-in-Time Recovery），简称 PITR。



3. Python

3.1 Python 基础 (Python Foundation)

3.1.1 Python 命名规范

Python 之父 Guido 推荐的命名规范, Python 模块、类、函数等命名示例如表3.1所示。对类名使用大写字母开头的单词 (如 `CapWords`, 即 Pascal 风格), 但是模块名应该用小写加下划线的方式 (如 `lower_with_under.py`)。尽管已经有很多现存的模块使用类似于 `CapWords.py` 这样的命名, 但现在已经不鼓励这样做, 因为如果模块名碰巧和类名一致, 这会让人困扰¹。

3.1.2 为什么要使用类

在编写 Python 的过程中, 突然发现很多模块是直接使用的方法, 而没有在模块中定义类, 突然想起 Python 既然用方法都可以完成的任务, 为什么要使用类呢? 不用类可以吗? 带着这个疑问, 查找了一番资料, 发现本质的区别在于 FP(Functional Pro-

¹https://zh-google-styleguide.readthedocs.io/en/latest/google-python-styleguide/python_style_rules/

Table 3.1: Python 命名规范

Type	Public	Internal
Modules	lower_with_under	_lower_with_under
Packages	lower_with_under	
Classes	CapWords	_CapWords
Exceptions	CapWords	
Functions	lower_with_under()	_lower_with_under()
Global/Class Constants	CAPS_WITH_UNDER	_CAPS_WITH_UNDER
Global/Class Variables	lower_with_under	_lower_with_under
Instance Variables	lower_with_under	_lower_with_under (protected) or __lower_with_under (private)
Method Names	lower_with_under()	_lower_with_under() (protected) or __lower_with_under() (private)
Function/Method	Parameters	lower_with_under
Local Variables	lower_with_under	

gramming) 和 OOP(Object-oriented Programming), 仿佛又进入了一个圣战领域。Python 既可以进行函数式编程, 也可以进行面向对象编程, 得分析具体的使用场景。

3.1.3 str 与 bytes

Python 3 中最重要的新特性可能就是将文本 (text) 和二进制数据做了更清晰的区分。文本总是用 unicode 进行编码, 以 str 类型表示; 而二进制数据以 bytes 类型表示。在 python3 中, 不能以任何隐式方式将 str 和 bytes 类型二者混合使用。不可以将 str 和 bytes 类型进行拼接, 不能在 str 中搜索 bytes 数据 (反之亦然), 也不能将 str 作为参数传入需要 bytes 类型参数的函数 (反之亦然)。只有在需要将 string 编码 (encode) 成 byte 的时候, 比如: 通过网络传输数据; 或者需要将 byte 解码 (decode) 成 string 的时候, 我们才会关注 string 和 byte 的区别。

```
1 # convert bytes for network transfer
2 data = bytes(encode_data_bytes,'utf8')
3 req = urllib.request.Request(url=url,data = data,headers = headers
    ,method='POST')
```

Python3 中, 在字符引号前加 'b', 明确表示这是一个 bytes 类型的对象, 实际上它就是一组二进制字节序列组成的数据, bytes 类型可以是 ASCII 范围内的字符和其它十六进制形式的字符数据, 但不能用中文等非 ASCII 字符表示。

3.1.4 序列化和反序列化

再将 Python 对象序列化到 SSDB 时, 直接保存对象即可, 不用转换为 Json 字符串后再保存, 否则取出的数据有双斜杠转义, 需要调取 2 次 Json 解析。Json 解析如下代码片段所示:

```
1 # ast(Abstract Syntax Trees) 是 python 中的一个模块
2 # 分析 python 的抽象语法树来对 python 的代码进行分析和修改
3 book_object = ast.literal_eval(book_text_str)
```

3.1.5 反反爬

在爬取豆瓣的过程中, 爬取一段时间后, 总是出现 404 错误。尝试添加 HTTP 请求头, 模拟浏览器的请求。但是在添加请求头之后, 没有产生预期的效果, 所以采用

Charles 工具来验证请求头。首先在代码中设置 Charles 代理：

```
1 proxy_support = request.ProxyHandler({'https': 'localhost:8888'})
2 opener = urllib.request.build_opener(proxy_support)
3 urllib.request.install_opener(opener)
4 req = urllib.request.Request(url)
```

在使用 HTTPS 代理时, 如果没有设置 https 代理解析, Charles 会提示 SSL Proxying not enabled for this host: enable in Proxy Settings, SSL locations, 如图3.1所示。

Name	Value
Host	https://api.douban.com:443
Path	/
Notes	SSL Proxying not enabled for this host: enable in Proxy Settings, SSL locations
▼ Requests	1
Completed	0
Incomplete	1
Failed	0
DNS	1
Connects	1
SSL Handshakes	0

Figure 3.1: Charles SSL 未设置代理提示

需要 Charles 设置 SSL 解析。会提示首先安装 CA SSL 证书, 其次在代理设置中启用 SSL 代理, 设置如图3.2所示。

设置完毕后, 在每次请求时, 可以在 Charles 中捕捉到请求, 并查看应用实际的请求头。最终发现是由于设置的 Agent 请求头未生效导致。豆瓣对单个 IP 爬取数据的频率有限制, 只能降低请求对频率来获取数据。最近发现可以生成一个代理的 IP 池, 使用免费的代理来突破爬取速度的限制, 经过比较, 选择了 proxy_pool²

3.1.6 调试

在 Visual Studio 中新增条件断点如图3.3所示。

scrapy_urls 返回爬取网页的链接列表, list 数据类型, 如果列表为空, 不保存。

²https://github.com/jhao104/proxy_pool

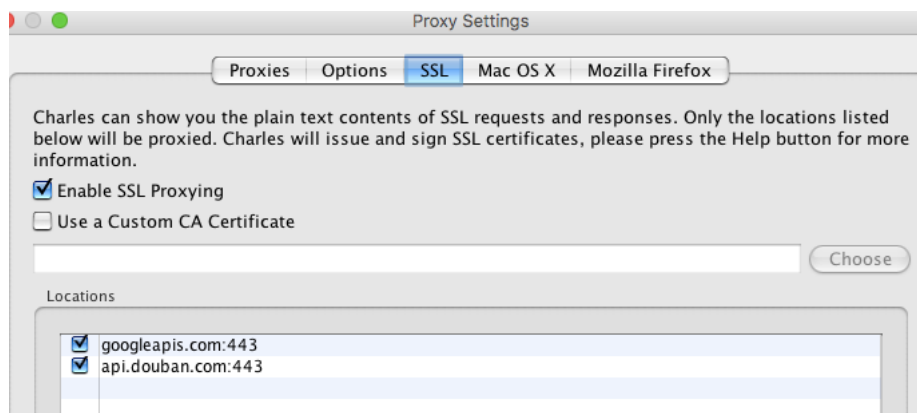


Figure 3.2: Charles 设置解析网站

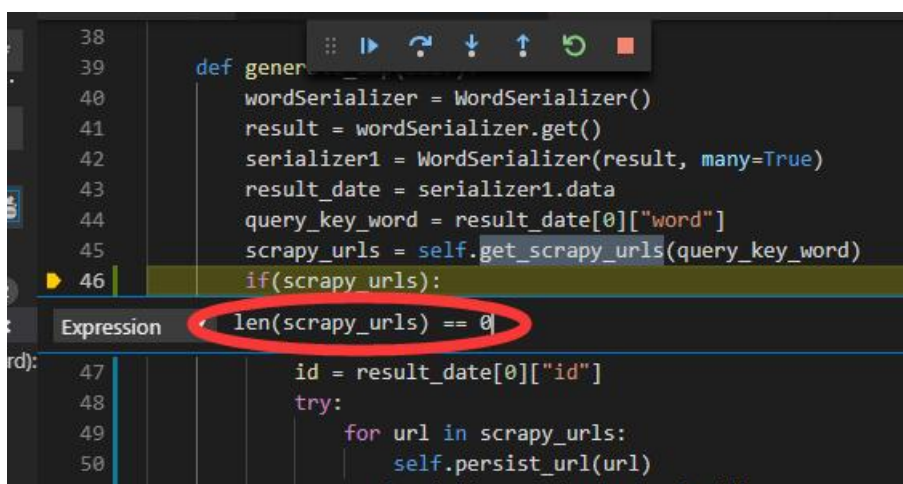


Figure 3.3: Visual Studio Code 条件断点

3.1.7 Unable to write settings

在使用 Visual Studio Code 开发时，由于 setting 文件由版本管理工具 Git 管理时冲突，导致无法正确读取和改写配置文件。

3.1.8 pipenv

Pipenv 附带包管理和虚拟环境支持，因此你可以使用一个工具来安装、卸载、跟踪和记录依赖性，并创建、使用和组织你的虚拟环境。你的应用程序可能依赖于某个特定版本的库，而那个库可能依赖于另一个特定版本的库，这些依赖关系如海龟般堆叠起来。当你的应用程序使用的两个库有冲突的依赖关系时，你的情况会变得很艰难。Pipenv 希望通过在一个名为 Pipfile.lock 的文件中跟踪应用程序相互依赖关系树来减轻这种痛苦。Pipfile.lock 还会验证生产中是否使用了正确版本的依赖关系。pipenv 的 locking 环节真是慢，使用下面的命令直接跳过 locking 环节：

```
1 pipenv install psycpg2 --skip-lock
```

在没有关联 Python 路径的情况下指定 Python 路径：

```
1 pipenv install --python /usr/bin/python3
```

3.1.9 Scrapy

Scrapy 需要依赖 Twisted 模块，Twisted 模块需要依赖 Visual C++³模块，但是进入网站却貌似没有找到独立模块的下载地址，安装 Visual Studio 不太现实，几个 GB。这个网站⁴已经有集成的包，下载集成好的包，直接离线安装即可。

```
1 # 离线安装 Twisted
2 pip3 install Twisted-18.9.0-cp37-cp37m-win_amd64.whl
3 # 安装 Scrapy
4 pip3 install scrapy
```

³下载地址 2019 年 1 月可用，链接为：<https://support.microsoft.com/en-us/help/2977003/the-latest-supported-visual-c-downloads>

⁴<https://www.lfd.uci.edu/~gohlke/pythonlibs/#twisted>

代理池 (Proxy Pool)

一般限制了爬虫的网站，在使用同一个 IP 在一段时间内发送了超过规定数量的请求，或者调取速度过于频繁，会立即被加入黑名单，封禁一段时间 (豆瓣一般是 24 小时)，所以如果需要大规模抓取数据，针对此种反爬手段就是建立一个代理集合，也就是代理池，请求通过分布在全球各地的代理机器转发。HTTP 代理池使用的是开源项目，新代理 IP 的抓取与存储，代理 IP 有效性的维护与检测，代理 IP 库对外的获取接口皆统一由开源组件来管理了。应用中需要做的就是调取对应代理服务的接口，获取一个随机的代理 IP。在 Scrapy 中定义一个代理 Middleware 如下代码片段所示：

```
1 class ProxyMiddleware(object):  
2  
3     def process_request(self, request, spider):  
4         proxy_address = self.get_random_proxy()  
5         request.meta["proxy"]="http://" + proxy_address
```

设置代理后，每次发送请求都会重新设置新的代理 IP，不用担心多个请求发送时会被封 IP，所以在使用代理的前提下可以在 Scrapy 爬取列表中同时设置多个请求 URL，当需要大批量抓取数据时，可以调整请求 URL 的列表，中间的调度，可以交给 Scrapy 来处理，Scrapy 可以方便的发送大规模的请求，而不需要费精力手动去管理线程，接收请求等，抓取到数据后，即可自动放入 Pipeline，可以理解为一个处理数据流的管道或者队列，在 IP 池足够多时，抓取的效率是非常可观的，前提是服务端能够承受大批量请求的压力。

优化 (Optimization)

提高并发数量，调整 settings 的 CONCURRENT_REQUESTS 参数。将下载延迟设为 0，这时需要相应的防 ban 措施，一般使用 user agent 轮转，构建 user agent 池，轮流选择其中之一来作为 user agent。设置 DOWNLOAD_DELAY 为 0，此项配置会导致服务器会在同一时刻接收到大规模集中的访问，对服务器的影响较大，相当于短时间里增大服务器负载。当代理地址响应较慢时，可以适当调低 DOWNLOAD_TIMEOUT 参数，如果不设置默认是 180s，那么应用会等待较长时间，会影响爬取当效率。明智的做法时快速失败，快速重试，第一个地址响应较慢，可以快速的使用另一个地址进行尝试。

Json 序列化

在 Scrapy 爬取内容后, 使用 ScrapyJSONEncoder 来转化为 Json, 再编码为 bytes 方式传输。ScrapyJSONEncode 转化为 Json:

```
1 _encoder = ScrapyJSONEncoder()
2 encode_result = _encoder.encode(data)
```

转化为 bytes:

```
1 encode_data_bytes = urllib.parse.quote_plus(encode_result)
2 data = bytes(encode_data_bytes,'utf8')
3 req = Request(url=url,data = data,headers = headers,method='POST')
4 response = urllib.request.urlopen(req).read()
5 response_text = str(response,'utf-8')
```

3.1.10 常见问题

No module named _sqlite3

编译时没有包含 sqlite, 输入如下命令安装:

```
1 yum -y install sqlite-devel
```

重新配置 (Re-configure), 重新编译 (Re-compiled)Python3:

```
1 # 重新执行 configure
2 ./configure --enable-loadable-sqlite-extensions
3 # 编译
4 make
5 # 安装
6 make install
```

callback 不调用

是因为启用了 allowed_domains 设置的缘故。allowed_domains 用于设置过滤爬取的域名, 在插件 OffsiteMiddleware 启用的情况下 (默认是启用的), 不在此允许范围

内的域名就会被过滤，因而不会进行爬取。但对于 `start_urls` 里的起始爬取页面，它不过滤，仅过滤起始爬取页面以外的页面。

3.1.11 Google Spider

Google 书籍链接的抓取地址：<https://www.googleapis.com/books/v1/volumes?q=ab>，其中 `q` 表示查询的关键字。为了数据爬取，构建一个公共的词库，目前使用清华大学的开放中文词库⁵、OmegaWiki⁶。将词库导入 Postgresql：

```
1 # 登陆 dolphin 数据库
2 psql -h 127.0.0.1 -p 5432 -d dolphin -U postgres
3 COPY fund_words_lib(word,remark) from '/Users/dolphin/Downloads/
   def_eng.csv' WITH CSV HEADER;
4 # 导入词库
5 COPY words(word) from '/Users/dolphin/source/pydolphin-service/
   dolphin/tool/file.csv' WITH CSV HEADER;
```

在抓取 Google API 数据时，在浏览器和终端可以请求到数据，但是 Scrapy 的下载器却不能请求到数据，是由于浏览器和终端默认使用的 Google API 的 IPv6 地址，而 Scrapy 目前未支持 IPv6 导致，IPv4 无法直接访问 Google 服务⁷。Google Book API⁸支持翻页，每页最大 40 条数据。

xpath

XPath 为 XML 路径语言 (XML Path Language)，它是一种用来确定 XML 文档中某部分位置的语言。简单示例如下：

```
1 sites = sel.xpath('//*[@id="detail_bullets_id"]/table/tr/td/div/ul
   /li')
```

以上表达式过滤出亚马逊商品的详细列表信息，双斜杠加星号表示选取文档中的所有元素，@ 表示选取属性，需要加上中括号。

⁵<http://thuoc1.thunlp.org/sendMessage>

⁶<http://www.omegawiki.org>

⁷<https://github.com/scrapy/scrapy/issues/3528>

⁸<https://developers.google.com/books/docs/v1/reference/volumes/list?hl=zh-CN>

3.1.12 api

启动爬虫接口服务端的命令如下：

```
1 # 启动 API 应用，服务端监听端口为 9000
2 /usr/local/bin/python3 manage.py runserver 0.0.0.0:9000
```

POST 请求接口：

```
1 curl -H "Content-Type:application/json" -X POST --data '{"index": "
   true"}' http://localhost:8000/spider/api/book
```

body 传送 json 数据：

```
1 curl -XPOST -H' Content-Type: application/json' http://localhost
   :8000/spider/api/book -d@temp.json
2 python3 manage.py runserver 0.0.0.0:9000
```

服务端接收请求：

```
1 location ~ /spider/api/ {
2     # 匹配爬虫服务 URL
3     # 匹配符合以后，停止往下搜索正则，采用这一条
4     proxy_pass http://spider-service;
5 }
```

JsonResponse

第一个参数，data 应该是一个字典类型，当 safe 这个参数被设置为：False，那 data 可以填入任何能被转换为 JSON 格式的对象，比如 list, tuple, set。默认的 safe 参数是 True。如果你传入的 data 数据类型不是字典类型，那么它就会抛出 TypeError 的异常。在返回 Json 响应的时候，一般不会直接返回查询出来的结果，要进行一次简单的标准化封装，返回如下格式的数据：

```
1 {
2     "code": "20000",
```



```
3     "data": {},
4     "message": "success"
5 }
```

通过新增自定义响应 CustomJsonResponse，封装成标准化的响应结果，效果如图3.4所示。

```
D:\project\source\pydolphin-service>http http://localhost:8000/spider/api/v1/word
HTTP/1.1 200 OK
Allow: GET, HEAD, OPTIONS
Content-Length: 185
Content-Type: application/json
Date: Thu, 10 Jan 2019 05:15:14 GMT
Server: WSGIServer/0.2 CPython/3.7.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
  "code": "20000",
  "data": [
    {
      "id": "24923",
      "remark": "An entry in a thesaurus or dictionary that is associated with a theme",
      "state": "0",
      "word": "is part of theme"
    }
  ],
  "desc": "get word success"
}
```

Figure 3.4: Django 自定义标准化 Json 响应

http http://localhost:8000/spider/api/v1/word

'utf8' codec can't decode byte 0x8b in position 1: invalid start byte

由于默认请求头包含 gzip 格式，服务端在返回的时候，对消息体进行了压缩。客户端发送 Accept-Encoding 请求头表示接收的内容类型。Python 默认不自动解压，无法识别，所以出现此问题。

```
1 'Accept-Encoding': 'gzip, deflate, br',
```

去掉 gzip 压缩类型即可。

3.2 Django

3.2.1 使用事务 (Using Transaction)

在爬取数据获取爬取链接时，由于同步运行有多个客户端，为了避免客户端获取到相同的链接，需要在查询接口中开启事务来避免此种情况。

```
1 # Avoid multi spider get the same key word
2 # Make each query atomic
3 # Pay attention the performance issue by transaction
4 @transaction.atomic
5 def get(self,request):
6     serializer = WordSerializer()
7     result = serializer.get()
8     serializer1 = WordSerializer(result, many=True)
9     return CustomJsonResponse(data=serializer1.data, code="20000"
        , desc='get word success' )
```

3.3 日志中心 (Logging Center)

根据 Google Trend 的信息显示，ELK Stack 已经成为目前最流行的集中式日志解决方案。ELK 的整体流程如图3.9所示。logstash 数据收集引擎。它支持动态的从各种数据源搜集数据，并对数据进行过滤、分析、丰富、统一格式等操作，然后存储到用户指定的位置。Logstash 项目诞生于 2009 年 8 月 2 日。其作者是世界著名的运维工程师乔丹西塞 (JordanSissel)，乔丹西塞当时是著名虚拟主机托管商 DreamHost 的员工，还发布过非常棒的软件打包工具 fpm，并主办着一一年一度的 sysadmin advent calendar(advent calendar 文化源自基督教氛围浓厚的 Perl 社区，在每年圣诞来临的 12 月举办，从 12 月 1 日起至 12 月 24 日止，每天发布一篇小短文介绍主题相关技术)。logstash 是 jvm 跑的，资源消耗比较大，启动一个 logstash 就需要消耗 500M 左右的内存，而 filebeat 只需要 10 来 M 内存资源。Filebeat 轻量型日志采集器。当您要面对成百上千、甚至成千上万的服务器、虚拟机和容器生成的日志时，Filebeat 将为您提供一种轻量型方法，用于转发和汇总日志与文件，让简单的事情不再繁杂。查看 Elastic 配置：

```
1 curl -XGET -H "Content-Type: application/json" http://localhost
```

```
:9200/_all/_settings|jq '.'
```

修改 Elastic 配置:

```
1 curl -XPUT -H "Content-Type: application/json" http://localhost
   :9200/_all/_settings
2 -d '{"index.blocks.read_only_allow_delete": null}'
```

对日志统一格式定义如下:

```
1 {
2   "date": "yyyy - MM - dd HH: mm: ss ",
3   "ip/host": "192.168.1.12",
4   "pid": "1211",
5   "topic": "track"
6   "type": "error",
7   "tag": "redis connection refused",
8   "platform": "java/go/php",
9   "level": "info/warn/error",
10  "app": "appName",
11  "module": "com.youzan.somemodule",
12  "detail": "any things you want here"
13 }
```

3.4 配置管理中心 (Configuration Management Center)

随着程序功能的日益复杂, 程序的配置日益增多: 各种功能的开关、参数的配置、服务器的地址。并且有对配置的各类要求, 配置修改后实时生效, 灰度发布, 分环境、分集群管理配置, 完善的权限、审核机制。并且随着采用分布式的开发模式, 项目之间的相互引用随着服务的不断增多, 相互之间的调用复杂度成不断升高, 因此需要引用配置中心治理。常用的配置管理工具有:

etcd 是由 CoreOS 开发并维护的, 灵感来自于 ZooKeeper 和 Doozer, 它使用 Go 语言编写, 并通过 Raft 一致性算法处理日志复制以保证强一致性。Raft 是一个来自

Table 3.2: 配置管理工具

工具名称	开发语言	备注
etcd	Go	分布式键值存储，用于共享配置和服务发现
consul	Go	一款服务发现和配置的工具
Doozer	Go	分布式的一致性数据存储服务

Stanford 的新的一致性算法。

Table 3.3: 配置管理工具

工具名称	开发语言	备注
etcd	Go	分布式键值存储，用于共享配置和服务发现
consul	Go	一款服务发现和配置的工具
Doozer	Go	分布式的一致性数据存储服务

etcd 是由 CoreOS 开发并维护的，灵感来自于 ZooKeeper 和 Doozer，它使用 Go 语言编写，并通过 Raft 一致性算法处理日志复制以保证强一致性。Raft 是一个来自 Stanford 的新的一致性算法。

```
restTemplate = {RestTemplate@13914}
  messageConverters = {ArrayList@13938} size = 7
  errorHandler = {DefaultResponseErrorHandler@13939}
  uriTemplateHandler = {DefaultUriTemplateHandler@13940}
  headersExtractor = {RestTemplate$HeadersExtractor@13941}
  interceptors = {ArrayList@13942} size = 0
  logger = {SLF4JLocationAwareLog@13943}
  requestFactory = {SimpleClientHttpRequestFactory@13944}
    proxy = null
    bufferRequestBody = true
    chunkSize = 4096
    connectTimeout = 1000
    readTimeout = 1000
    outputStreaming = true
    taskExecutor = null
```

Figure 3.5: RestTemplate 默认超时时间

clnt_ip	state	user_name	COUNT(*)
::ffff:0:0:0:0:0:0:0:1.1.1	IDLE	USER_NAME_NEW	45
::ffff:0:0:0:0:0:0:0:1.1.1	IDLE	USER_NAME_BASIC_NEW	25
::ffff:0:0:0:0:0:0:0:1.1.1	IDLE	USER_NAME_DOG_NEW	129
::ffff:0:0:0:0:0:0:0:1.1.1	IDLE	USER_NAME_NEW	7
::ffff:0:0:0:0:0:0:0:1.1.1	IDLE	USER_NAME_NEW	86
::ffff:0:0:0:0:0:0:0:1.1.1	IDLE	USER_NAME_NEW	3
::ffff:0:0:0:0:0:0:0:1.1.23	IDLE	USER_NAME_NEW	4
::ffff:0:0:0:0:0:0:0:1.1.23	IDLE	USER_NAME_DOG_NEW	20
::ffff:0:0:0:0:0:0:0:1.1.23	IDLE	USER_NAME_NEW	4
::ffff:0:0:0:0:0:0:0:1.1.1	IDLE	USER_NAME_DOG	5
::ffff:0:0:0:0:0:0:0:1.1.1	IDLE	USER_NAME_NEW	1
::ffff:0:0:0:0:0:0:0:1.1.1	ACTIVE	USER_NAME_NEW	12
::ffff:0:0:0:0:0:0:0:0.1	ACTIVE	USER_NAME_NEW	1

Figure 3.6: DB 当前连接会话统计

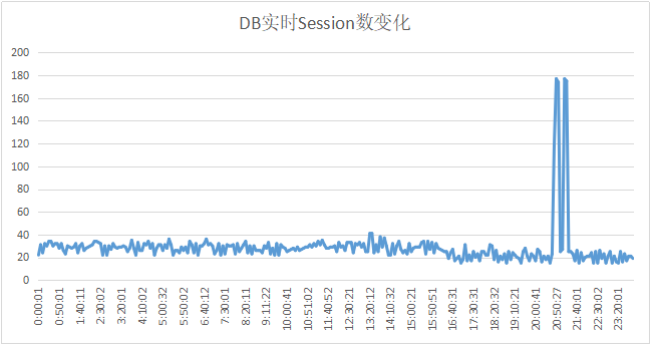


Figure 3.7: DB 历史会话数统计

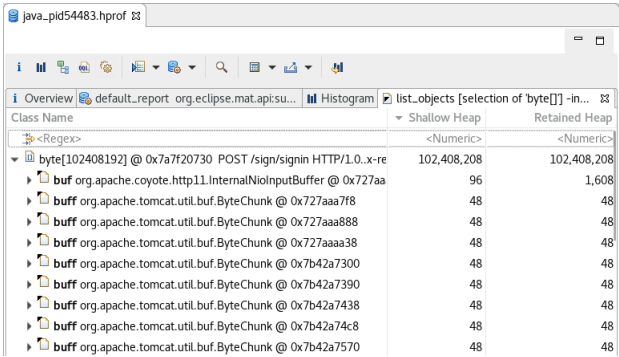


Figure 3.8: Java 内存对象分析

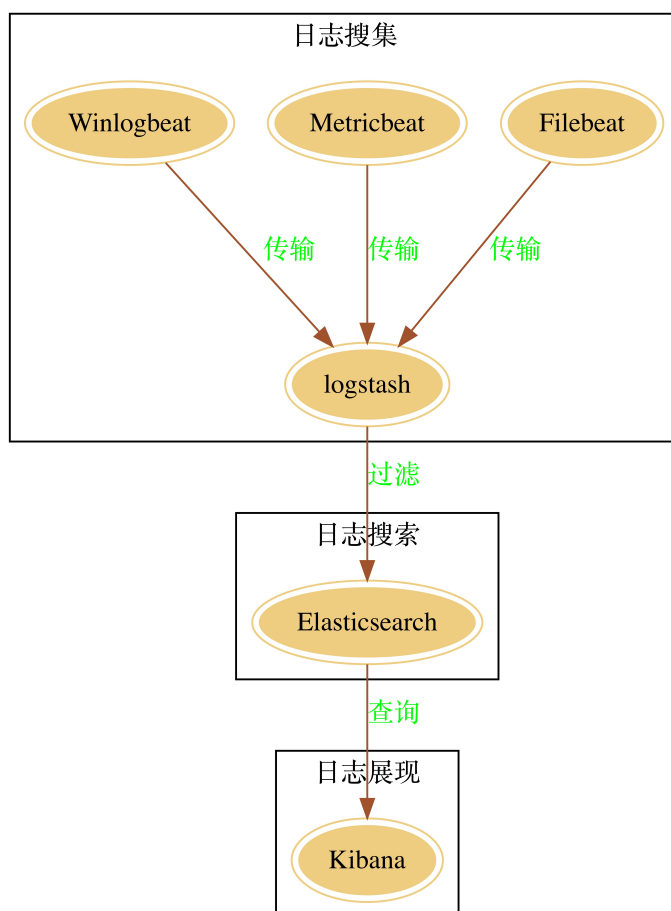


Figure 3.9: ELK 流程

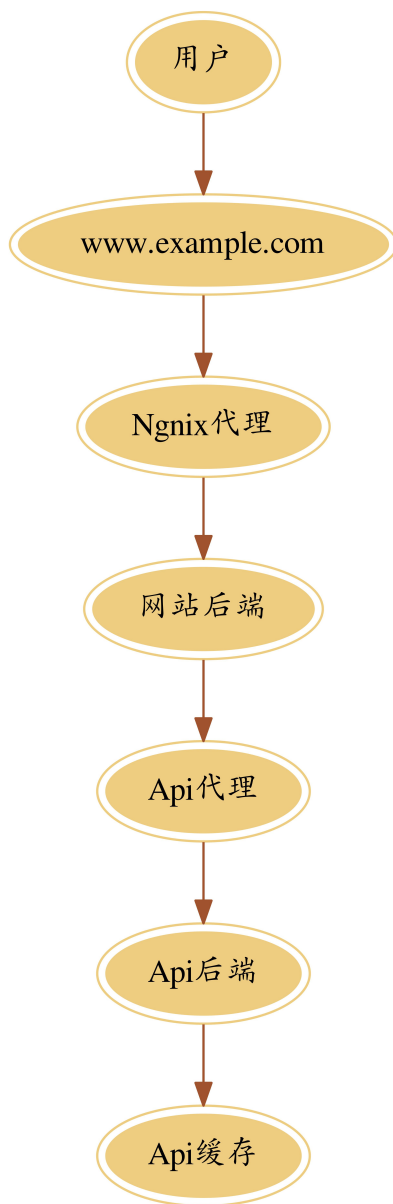


Figure 3.10: 接口数据调用流程

#	Hits ▼	Visitors ↕	Bandwidth ↕	Method ↕	Protocol ↕	Data ↕
	2,119,878 Max: 1,925,552 Min: 1	92,426 Max: 3,110 Min: 1	0 Byte Max: 0 Byte Min: 0 Byte			35,225 Total
1	1,925,552 (90.83%)	1,180 (1.28%)	0 Byte (0.00%)	POST	HTTP/1.1	/html/query/collect.html
2	15,670 (0.74%)	3,110 (3.36%)	0 Byte (0.00%)	GET	HTTP/1.1	/
3	4,616 (0.22%)	441 (0.48%)	0 Byte (0.00%)	GET	HTTP/1.1	/html/query/getAllType.html
4	3,948 (0.19%)	569 (0.62%)	0 Byte (0.00%)	POST	HTTP/1.1	/html/query/querySensitive.html
5	3,185 (0.15%)	1,667 (1.80%)	0 Byte (0.00%)	GET	HTTP/1.1	/resource/default/js/layer/skin/default/layer
6	3,037 (0.14%)	1,630 (1.76%)	0 Byte (0.00%)	GET	HTTP/1.1	/resource/static/css/ui2.css?2013032917
7	2,651 (0.13%)	1,141 (1.23%)	0 Byte (0.00%)	GET	HTTP/1.1	/html/query/collect.html

Figure 3.11: 访问 URL 统计

#	Hits ▼	Visitors ↕	Bandwidth ↕	Hostname ↕	Data ↕
	2,338,695 Max: 483,972 Min: 1	12,508 Max: 6 Min: 1	0 Byte Max: 0 Byte Min: 0 Byte		6,782 Total
1	483,972 (20.69%)	0 (0.00%)	0 Byte (0.00%)	Name or service not known	139.224.64.123
2	482,241 (20.62%)	0 (0.00%)	0 Byte (0.00%)	Name or service not known	139.224.54.163
3	323,217 (13.82%)	0 (0.00%)	0 Byte (0.00%)	Name or service not known	120.27.160.48
4	21,600 (0.92%)	3 (0.02%)	0 Byte (0.00%)	Name or service not known	39.155.185.29
5	11,163 (0.48%)	3 (0.02%)	0 Byte (0.00%)	Name or service not known	222.182.185.102
6	10,104 (0.43%)	1 (0.01%)	0 Byte (0.00%)	Name or service not known	111.202.154.66
7	9,513 (0.41%)	2 (0.02%)	0 Byte (0.00%)	Name or service not known	222.182.185.108

Figure 3.12: 根据 IP 统计

```
[root@localhost logs]# netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}'
TIME_WAIT 79
CLOSE_WAIT 167
FIN_WAIT2 178
ESTABLISHED 706
SYN_RECV 23
LAST_ACK 15
```

Figure 3.13: 查看服务器并发示例

#	Hits ▼	Visitors ↕	Bandwidth ↕	Method ↕	Protocol ↕	Data ↕
	2,119,878 Max: 1,925,552 Min: 1	92,426 Max: 3,110 Min: 1	0 Byte Max: 0 Byte Min: 0 Byte			35,225 Total
1	1,925,552 (90.83%)	1,180 (1.28%)	0 Byte (0.00%)	POST	HTTP/1.1	/html/query/collect.html
2	15,670 (0.74%)	3,110 (3.36%)	0 Byte (0.00%)	GET	HTTP/1.1	/
3	4,616 (0.22%)	441 (0.48%)	0 Byte (0.00%)	GET	HTTP/1.1	/html/query/getAllType.html
4	3,948 (0.19%)	569 (0.62%)	0 Byte (0.00%)	POST	HTTP/1.1	/html/query/querySensitive.html
5	3,185 (0.15%)	1,667 (1.80%)	0 Byte (0.00%)	GET	HTTP/1.1	/resource/default/js/layer/skin/default/layer
6	3,037 (0.14%)	1,630 (1.76%)	0 Byte (0.00%)	GET	HTTP/1.1	/resource/static/css/ui2.css?2013032917
7	2,651 (0.13%)	1,141 (1.23%)	0 Byte (0.00%)	GET	HTTP/1.1	/html/query/collect.html

Figure 3.14: 访问 URL 统计

#	Hits ▼	Visitors ↕	Bandwidth ↕	Hostname ↕	Data ↕
	2,338,695 Max: 483,972 Min: 1	12,508 Max: 6 Min: 1	0 Byte Max: 0 Byte Min: 0 Byte		6,782 Total
1	483,972 (20.69%)	0 (0.00%)	0 Byte (0.00%)	Name or service not known	139.224.64.123
2	482,241 (20.62%)	0 (0.00%)	0 Byte (0.00%)	Name or service not known	139.224.54.163
3	323,217 (13.82%)	0 (0.00%)	0 Byte (0.00%)	Name or service not known	120.27.160.48
4	21,600 (0.92%)	3 (0.02%)	0 Byte (0.00%)	Name or service not known	39.155.185.29
5	11,163 (0.48%)	3 (0.02%)	0 Byte (0.00%)	Name or service not known	222.182.185.102
6	10,104 (0.43%)	1 (0.01%)	0 Byte (0.00%)	Name or service not known	111.202.154.66
7	9,513 (0.41%)	2 (0.02%)	0 Byte (0.00%)	Name or service not known	222.182.185.108

Figure 3.15: 根据 IP 统计

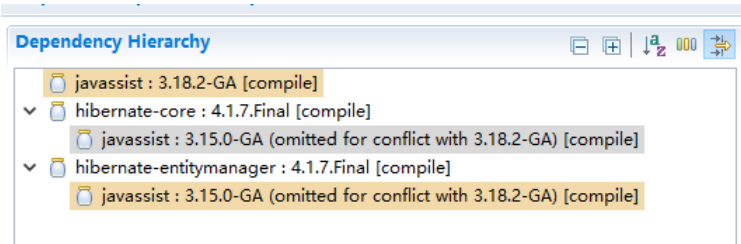


Figure 3.16: 依赖冲突

4. MQ(Message Queue)

4.1 Apache Kafka

4.1.1 Producer

创建生产者提示 `kafka.errors.NoBrokersAvailable` exception。此时需要配置服务端 `server.properties`:

```
1 # Hostname and port the broker will
2 # advertise to producers and consumers.
3 # If not set, it uses the value for "listeners" if configured.
4 # Otherwise, it will use the value returned from
5 # java.net.InetAddress.getCanonicalHostName().
6 # 这里配置的地址要保证生产者和消费者能够直接访问
7 advertised.listeners=PLAINTEXT://10.142.0.2:9092
```

`advertised` 监听的配置是有讲究的，掉这个坑里面整整一天。这个配置是生产者和

消费者需要使用的, 开始的时候配置的是内网的地址, 这就出问题了, 生产者拿到这个配置后一直向内网 IP 发送消息, 由于 Kafka 部署在 Google 云上, 机器在美国, 根本不在同一个局域网, 肯定是无法发送成功的, 导致生产者一直超时, 无法获取到元数据。所以这里需要配置公网的 IP 地址, 调整后即可解决问题。如果没有设置, 将会使用 listeners 的配置, 如果 listeners 也没有配置, 将使用 `java.net.InetAddress.getCanonicalHostName()` 来获取这个 hostname 和 port。将 localhost 调整为 IP 后, 服务端消费时提示 `Broker may not be available`, 是由于为了支持外部访问 Kafka, `server.properties` 文件中指定绑定 IP, 此时生产消费脚本也需要通过 IP 来进行。advertised.listeners 为 Kafka 0.9.x 以后的版本新增配置, 原来的 `advertise.host.name` 和 `port` 配置不建议再使用。同时在 Python 初始化的时候添加 API 版本参数。

```
1 producer = KafkaProducer(  
2     bootstrap_servers=['mq-server:9092'],  
3     api_version_auto_timeout_ms=10000,  
4     # fix no broker available problem  
5     api_version = (0, 10)  
6 )
```

出现此错误 `KafkaTimeoutError('Failed to update metadata after 60.0 secs.')` 是由于 Python 的 Kafka 客户端版本不匹配, 服务端安装的是最新的 2.x 版本, 而客户端最高支持 1.1 版本, 所以会出现此问题, 降低服务端版本即可解决此问题。使用命令创建 Topic:

```
1 bin/kafka-topics.sh --create --zookeeper localhost:2181 --  
    replication-factor 1 --partitions 1 --topic dolphin-test
```

安装完成后, 可以启动消费者, 再使用生产者脚本发送信息, 在消费端查看输出, 即可验证 Kafka:

```
1 # 消费 dolphin-test 主题消息  
2 bin/kafka-console-consumer.sh --bootstrap-server 10.142.0.2:9092  
    --topic dolphin-spider-google-book-bookinfo --from-  
    beginning  
3 # 生产 dolphin-test 主题消息
```

```
4 bin/kafka-console-producer.sh --broker-list 10.142.0.2:9092 --
    topic dolphin-test
```

测试 Kafka 消息生产与消费过程如图4.1所示，生产者向主题 dolphin-test 发送 Hello,Dolphin! 消息，消费者消费到此条消息：

```

[root@instance-1 kafka_2.11-2.1.0]# ls
bin config libs LICENSE logs nohup.out
[root@instance-1 kafka_2.11-2.1.0]# bin/kafka-console-producer.sh --broker-list 10.142.0.2:9092 --topic dolphin-test
Hello,dolphin!
Hello,my name is dolphin,sending message...
[root@instance-1 kafka_2.11-2.1.0]# bin/kafka-console-consumer.sh --zookeeper-quorum 10.142.0.2:2181 --topic dolphin-test
Hello,dolphin!
Hello,my name is dolphin,sending message...

```

Figure 4.1: Kafka 发送消息测试

4.1.2 Consumer

Kafka 消费者 Python 客户端如下代码片段所示 (不同 Kafka 版本略有区别)：

```

1 consumer = KafkaConsumer('dolphin-test',
2                             bootstrap_servers=['mq-server:9092'],
3                             group_id = "console-consumer-30308",
4                             consumer_timeout_ms=5000)
5
6 def consume_bookinfo(self):
7     while True:
8         for msg in self.consumer:
9             recv = "%s:%d:%d: key=%s value=%s" % (msg.topic, msg.
10                partition, msg.offset, msg.key, msg.value)
11             print(recv)

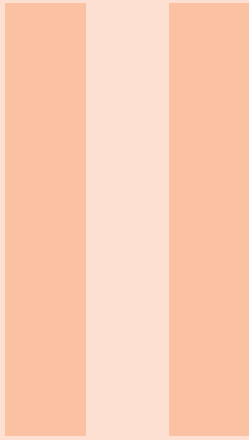
```

console-consumer-30308 表示消费控制台组的消息，可以使用命令将消息推送到控制台，在代码中使用客户端获取，调试消费着非常方便，参数 group_id 必须要指定。dolphin-test 是消息的主题。Java 版本的消费者客户端如下代码片段所示：

```
1 Properties props = new Properties();
2 props.put("bootstrap.servers", "mq-server:9092");
3 props.put("group.id", "console-consumer-30308");
4 props.put("enable.auto.commit", "true");
5 props.put("auto.commit.interval.ms", "1000");
6 props.put("key.deserializer", "org.apache.kafka.common.
    serialization.StringDeserializer");
7 props.put("value.deserializer", "org.apache.kafka.common.
    serialization.StringDeserializer");
8 KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props
    );
9 consumer.subscribe(Arrays.asList("dolphin-test", "dolphin-spider-
    google-book-bookinfo"));
10 while (true) {
11     ConsumerRecords<String, String> records = consumer.poll(100);
12     for(ConsumerRecord<String, String> record :records){
13         System.out.println("value:" + record.value());
14     }
15 }
```

group.id 是一个字符串，唯一标识一个 Consumer Group。多个消费者或消费者实例 (Consumer Instance)，它们共享一个公共的 ID，即 Group ID。组内的所有消费者协调在一起来消费订阅主题 (Subscribed Topics) 的所有分区 (Partition)。当然，每个分区只能由同一个消费组内的一个 Consumer 来消费。在 Consumer 中可以自定义 Group ID(不需要 Producer 生产数据时指定)，当 Consumer 消费数据后，Group ID 自动注册到服务端。使用如下命令查看当前 Group ID 有哪些 Topic：

```
1 bin/kafka-consumer-groups.sh --bootstrap-server mq-server:9092 --
    describe --group google-book
```



Tool

5 Widgets 69

- 5.1 Little Tool
- 5.2 Develop
- 5.3 WireGuard

6 Docker 79

- 6.1 Docker 基础



5. Widgets

5.1 Little Tool

5.1.1 traceroute

通过 traceroute 我们可以知道信息从计算机到互联网另一端的主机是走的路由路径。Traceroute 程序的设计是利用 ICMP 及 IP header 的 TTL (Time To Live) 栏位 (field)。首先, traceroute 送出一个 TTL 是 1 的 IP datagram (其实, 每次送出的为 3 个 40 字节的包, 包括源地址, 目的地址和包发出的时间标签) 到目的地, 当路径上的第一个路由器 (router) 收到这个 datagram 时, 它将 TTL 减 1。此时, TTL 变为 0 了, 所以该路由器会将此 datagram 丢掉, 并送回一个「ICMP time exceeded」消息 (包括发 IP 包的源地址, IP 包的所有内容及路由器的 IP 地址), traceroute 收到这个消息后, 便知道这个路由器存在于这个路径上, 接着 traceroute 再送出另一个 TTL 是 2 的 datagram, 发现第 2 个路由器……。traceroute 每次将送出的 datagram 的 TTL 加 1 来发现另一个路由器, 这个重复的动作一直持续到某个 datagram 抵达目的地。当 datagram 到达目的地后, 该主机并不会送回 ICMP time exceeded 消息, 因为它已是目的地了, 那么 traceroute 如何得知目的地到达了呢? Traceroute 在送出 UDP datagrams 到目的地时, 它

所选择送达的 port number 是一个一般应用程序都不会用的号码 (30000 以上), 所以当此 UDP datagram 到达目的地后该主机会送回一个「ICMP port unreachable」的消息, 而当 traceroute 收到这个消息时, 便知道目的地已经到达了。所以 traceroute 在 Server 端也是没有所谓的 Daemon 程式。Traceroute 提取发 ICMP TTL 到期消息设备的 IP 地址并作域名解析。每次, Traceroute 都打印出一系列数据, 包括所经过的路由设备的域名及 IP 地址, 三个包每次来回所花时间。查看 ipv6 的路由路径 (常用 IPv6 地址¹):

```
1 # Google 公共 IPv6 DNS 地址 1
2 # 2001:4860:4860::8844 (google-public-dns-a.google.com)
3 # Google 公共 IPv6 DNS 地址 2
4 # google-public-dns-a.google.com
5 traceroute6 2001:4860:4860::8888
```

5.1.2 dig

dig 命令最典型的用法就是查询单个主机的信息。dig is a flexible tool for interrogating DNS name servers.

```
1 # 查询 Google 的 IPv6 地址
2 dig www.google.com AAAA
3 # 查询 baidu 的 IPv6 地址
4 dig ipv6.baidu.com AAAA
5 # host 命令查看 IPv6 地址
6 host -t AAAA www.google.com
```

AAAA 表示查看对应域名的 IPv6 地址。也可以通过 ping6 命令查看对应的 IPv6 地址:

```
1 # 查询 baidu 的 IPv6 地址
2 ping6 ipv6.baidu.com
3 # 查看 Google API 的 IPv6 地址
4 # 2404:6800:4005:802::200a
```

¹<https://raw.githubusercontent.com/lenny1xx/ipv6-hosts/master/hosts>

```
5 ping6 https://www.googleapis.com
6 # Wireshark 过滤 IPv6 流量
7 ipv6 and ipv6.addr==2404:6800:4005:80b::200a
```

输出 PING6(56=40+8+8 bytes) local ip -> 2400:da00:2::29。local ip 是本机的 IPv6 地址。

5.1.3 ssh

使用如下命令动态代理：

```
1 # ssh 利用 Google 云主机动态代理
2 ssh -Nf -D 127.0.0.1:2000 google-cloud
3 # 链接到 192.168.1.11 的流量转发到 192.168.0.14
4 ssh -g -fN -L 8100:192.168.0.14:8100 192.168.1.11
```

服务端有大量拒绝连接输出，且部分网站无法正常访问，不知原因在何处。访问腾讯视频时，提示所在地区无法访问，可能是国外相应的服务封锁了部分地区的 IP。

5.2 Develop

5.2.1 Mybatis Aotogenerate

表模糊匹配

数据库中有一批表名称以 book 开头，如果需要生成这一批表的 POJO、Mapper，可以在配置文件中做如下配置：

```
1 <table tableName="book%"></table>
```

5.2.2 Medis

Redis 的客户端百花齐放，不同语言的客户端都有，质量参差不齐，选择起来也是一头雾水。偶然发现 Medis²客户端，开源、简洁，试用效果还可以。Medis 是基于 Electron, React, and Redux 开发的一款工具，设计是在 Mac 上使用，不过由于 Electron

²<https://github.com/luin/redis>

本身的跨平台特性，Windows 下也有相应的安装包³。Redis Desktop Manager 安装非常费劲，Mac 下的 binary 包需要收费，直接自己编译也相当麻烦的。需要注意的是，Medis 只支持 Redis 2.8 及以上版本，因为 SCAN 命令是 2.8 版本才引入的特性，它可以获取 key 而不会阻塞服务器，这个特性在生产环境是非常重要的。

5.2.3 Consul

Consul 是 HashiCorp 公司推出的开源工具，用于实现分布式系统的服务发现与配置。与其他分布式服务注册与发现的方案，比如 Airbnb 的 SmartStack 等相比，Consul 的方案更“一站式”，内置了服务注册与发现框架、分布一致性协议实现、健康检查、Key/Value 存储、多数据中心方案，不再需要依赖其他工具（比如 ZooKeeper 等）。Consul 用 Golang 实现，因此具有天然可移植性（支持 Linux、windows 和 Mac OS X）；安装包仅包含一个可执行文件，方便部署，与 Docker 等轻量级容器可无缝配合。

```
1 # 启动代理
2 ./consul agent -dev -ui -config-dir /Users/fox/bin/consul.d/
```

启动后，访问地址 <http://localhost:8500/ui> 即可。

5.3 WireGuard

5.3.1 WireGuard 配置

WireGuard 可能会合并进入 Linux 内核的，以其简单优雅的设计和实现得到了 Linus Torvalds 的首肯：

Can I just once again state my love for it and hope it gets merged soon? Maybe the code isn't perfect, but I've skimmed it, and compared to the horrors that are OpenVPN and IPSec, it's a work of art.

初次搭建 OpenVPN 时，最深刻的感受就是配置复杂。相对于 OpenVPN，WireGuard 的优势主要体现在：配置简单方便，速度更快。在 Mac 下，输入如下命令安装 WireGuard：

```
1 sudo brew install wireguard-tools
```

³<https://github.com/classfellow/redis/releases/tag/win>

生成公钥和私钥：

```
1 wg genkey | tee privatekey | wg pubkey > publickey
```

在服务器上，配置以下内容：

```
1 [Interface]
2 Address = 192.168.2.1/24
3 PrivateKey = <server's privatekey>
4 ListenPort = 51820
5
6 [Peer]
7 PublicKey = <client's publickey>
8 AllowedIPs = 192.168.2.2/32
```

使用 192.168.2.0/24 子网作为我们 VPN 的地址空间。对于每台计算机，您需要在此范围内选择一个唯一的地址 (192.168.0.1 到 192.168.0.254)，并使用 CIDR 表示法指定地址和子网。对于客户端，这是配置：

```
1 [Interface]
2 Address = 192.168.2.2/24
3 PrivateKey = <client's privatekey>
4 ListenPort = 51820
5
6 [Peer]
7 # 服务端的公钥
8 PublicKey = <server's publickey>
9 Endpoint = <server's ip>:51820
10
11 # WireGuard 全局生效
12 # 将所有流量都路由，相当于全局代理
13 AllowedIPs = 0.0.0.0/0
14
15 # AllowedIPs = 192.168.2.2/32
```

```
16 # This is for if you're behind a NAT and
17 # want the connection to be kept alive.
18 PersistentKeepalive = 25
```

配置完毕后，输入如下命令启动 Wireguard：

```
1 # 启动 Wireguard
2 wg-quick up wg0
3 # 停止 Wireguard
4 wg-quick down wg0
```

wg-quick 是 WireGuard 用来启动网络设备的 Bash 脚本。现在本地跟服务器已经在同一个内网上，可以彼此通信。但本地现在是无法通过服务器连接到外面的网络，如果需要通过服务器连接到外面的网络，要在服务器上设置流量转发和 NAT(Network Address Translation，网络地址转换) 才可以。

5.3.2 使用问题

5.3.3 可以连接，无法上网

启动 WireGuard 后，可以 ping 通百度，但是无法上网。使用如下 traceroute 命令跟踪：

```
1 traceroute -I www.baidu.com
```

I 参数强制使用 ICMP 包请求以及确认回应，即使用 ICMP Echo Request，Echo Reply 和 TTL-expired。在分析 traceroute 的结果时需要知道的是，如果路由封了 type 11 (TTL-expired)，中间的 router 全看不到，回显的是星号，但能看到 packet 到达了最后的 destination；如果封了 ICMP Echo Reply，中间的全能看到，最后的 destination 看不到。根据 traceroute 结果可知，发送的 ICMP 包的确经过 WireGuard 服务器到达了目标。使用 tcpdump 抓包：

```
1 tcpdump -i eth0 -A -s 0 'tcp port 80 and (((ip[2:2] - ((ip[0]&0xf)
    <<2)) - ((tcp[12]&0xf0)>>2)) != 0)'|grep "bilibili"
```

bilibili/duckduckgo/yandex 可以访问, baidu 不可以, 不知道什么原因。在配置的过程中, 发现不同节点的机器不能使用重叠的子网, 如果重叠, 后面配置的节点无法联网, 例如节点 A 设置了允许 192.168.2.1 至 192.168.2.255 的子网, CIDR 的写法是 192.168.2.0/24, 那么节点 B 的 IP 就不能落在此范围内, 如图 5.1 所示。客户端 IP 使用的是 192.168.3.* 子网, 而不是 192.168.2.* 子网。连接上之后, 可以直接访问 192.168.2.* 网段的节点, 至此公司的 PC、Google 云服务器和家里的 PC 组成了一个局域网, 只要通过公网 IP 连接到了 Wireguard(10.142.0.2), 在任何地点皆可以访问局域网的节点。

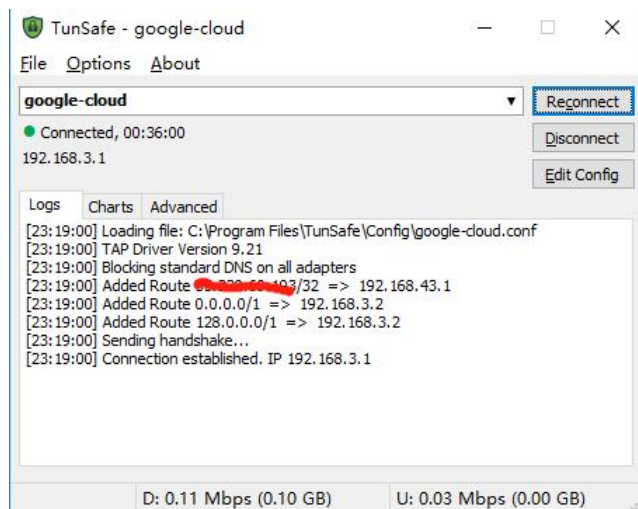


Figure 5.1: Wireguard Windows 客户端 tunsafe

5.3.4 应用

利用 Google Cloud 公网 IP 搭建 Wireguard 后, 可以形成一个虚拟的局域网, 同时可以利用此环境做许多事情, 当然任何事情首先得遵守相关规定。例如可以访问 Google, 查找资料、可以访问 Google 学术, 搜寻全球的论文、自己搭建集群环境、自己编写应用部署到 Google Cloud、访问局域网你们的设备、搭建自己的博客、申请自己的域名映射到 Google Cloud 博客服务器等等。有了 Wireguard 搭建的 VPN 服务, 还可以增加 Google Cloud 云主机的安全性, 对外只暴露特定端口即可, 其他应用可以直接通过内网 IP 进行访问。

ssh 登录

使用 Wireguard 连接后, Google Cloud 上的云虚拟机与本地的节点相当于在同一个局域网, 那么可以使用普通的 SSH 直接登录到 Google Cloud 远程虚拟机。更进一步, 可以将连接上 Wireguard 的所有节点当成是一个本地局域网, 不同的主机在不同的物理位置, 这样就可以更加方便的搭建集群环境。虽然由于物理主机像个太远造成的网络延时无法应用于实际的生产, 但是作为平时的学习研究还是可以接受的。

```
1 # 编辑 sshd 配置, 允许 root 用户登录 (不安全)
2 vim /etc/ssh/sshd_config
3 # 重新启动 ssh 守护进程
4 service sshd restart
5 # 利用内网 IP 登录到 Google Cloud 主机
6 ssh root@10.142.0.2
```




6. Docker

6.1 Docker 基础

6.1.1 为什么要用 Docker

好不容易配置部署了 redmine，结果后面换了一台机器，又重新部署一遍？其实绝大部都是配置的工作，繁琐又重复，其实我们没有办法也没有必要记住每一步操作步骤，先安装什么，安装好了再修改哪个文件的配置，重复的劳动没有任何意义。开发过程中会发现，许多工作都可以总结出来套路的，做过一遍之后，以后的内容都是高度重复。比如安装个软件，部署个应用，特别时多机部署，要保证一致性非常麻烦。还有经常会有问题：在我的电脑上好好的呀？以上这些问题，就是 Docker 要解决的问题，可大量减少花费在维护、配置不同系统的时间。常用命令：

```
1 # 查看当前运行的容器
2 docker ps
3 # 登陆 Docker 容器
4 docker exec -it f5605d02f9f5 bash
```

```
5 # 登陆 Docker 后, 登陆 MariaDB
6 mysql -h127.0.0.1 -uroot -p123456
```

6.1.2 Docker 安装

6.1.3 安装数据层

查看 MariaDB:

```
1 docker search mariadb
```

安装 MariaDB:

```
1 docker pull mariadb
```

安装时, 从国外镜像下载的速非常非常慢, 首先将下载源调整为国内的下载源。对于使用 systemd 的系统, 在/etc/docker/daemon.json 中写入如下内容 (不存在此文件新建即可):

```
1 {
2     "registry-mirrors": [
3         "https://registry.docker-cn.com"
4     ]
5 }
```

运行容器:

```
1 docker run --name MariaDB \
2     -p 3306:3306 \
3     -v /data/db/mariadb:/var/lib/mysql \
4     -e MYSQL_ROOT_PASSWORD=123456 \
5     -d mariadb
6 # 运行 Jenkins
7 docker run -p 8080:8080 \
8     -p 50000:50000 \
```

```
9 -v /usr/local/work/jenkins:/var/jenkins_home \  
10 --name j01 -idt jenkins
```

p 表示要自定义映射的端口 (port)，可以用 -p hostPort:containerPort。为了方便迁移数据库中的数据，可以通过挂载数据卷 (volume) 来实现。这样，数据库中的数据将保存在我们挂载的本地文件/data/db/mariadb 的上。我们可以迁移或者备份这个文件夹，来实现数据库迁移。一般一个自动生成的空数据库文件，大概有 100 多兆，而且这个文件夹中包含很多子文件，因此如果通过 SSH 或者 FTP 传输都需要比较长的时间，可以通过压缩打包来减少文件夹的容量。启动 MariaDB 镜像：

```
1 # 启动 MariaDB  
2 docker run --name mariadb -p 3306:3306 -e MYSQL_ROOT_PASSWORD=  
    password -d mariadb  
3  
4 # 启动 Jenkins  
5 docker run -p 8080:8080 -v /usr/local/work/jenkins:/var/  
    jenkins_home -t jenkins
```

启动 Jenkins 可能会出现如下错误：

```
1 touch: cannot touch '/var/jenkins_home/copy_reference_file.log' :  
    Permission denied  
2 Can not write to /var/jenkins_home/copy_reference_file.log. Wrong  
    volume permissions?
```

当映射本地数据卷时，/home/docker/jenkins 目录的拥有者为 root 用户，而容器中 jenkins user 的 uid 为 1000。需要调整目录权限：

```
1 sudo chown -R 1000:1000 /home/docker/jenkins
```

启动后就可以通过本地的 8080 端口登陆 Jenkin 了。

6.1.4 Docker 导入导出

查看名称：

```
1 docker ps
```

导出:

```
1 docker import -o MariaDB.tar MariaDB
```

导入:

```
1 docker export MariaDB.tar MariaDB
```



Network

7	HTTP	85
7.1	Fiddler	
7.2	Web Server	



7. HTTP

7.1 Fiddler

7.1.1 排查 Google Api 拒绝原因

在抓取 Google API 数据的过程中，部署到部分机器上提示 Connection was refused by other side: 111。原来是在本机调试的时候设置了代理，而部署到墙外服务器没有安装代理软件，其实根本不需要代理。去掉中间件 (middleware) 中的代理即可，移除如下代码片段：

```
1 proxy_address = '127.0.0.1:8888'  
2 request.meta["proxy"]="https://" + proxy_address
```

当然也有可能是服务端识别到了爬虫的请求头。在使用 urllib 库时，默认的请求头是：Python-urllib/3.7，很明显不是浏览器发出，服务器会认为是自动的 robot 发出此请求，无情予以封锁。Python 发出的 HTTP 请求头可以在 Fiddler 中查看：

同样的，可以通过 Fiddler 查看请求 Google API 的请求头，从而通过调整请求头，

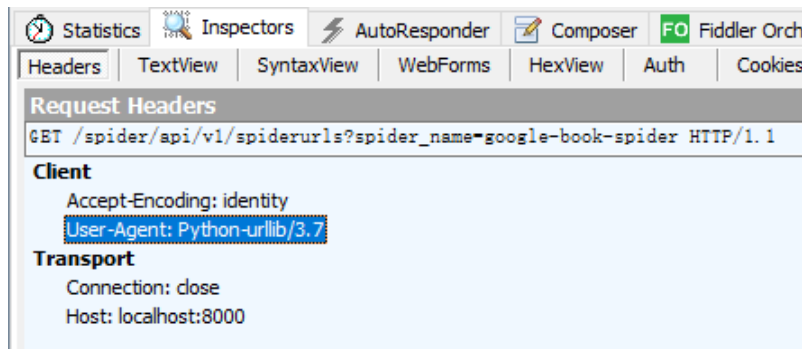


Figure 7.1: Fiddler 查看 Python HTTP 请求头

避免被服务端识别。有的服务器会检查 refer 请求头，只允许本站的链接，从而避免外链，避免其他网站链接嵌入从而消耗自身的宝贵服务器资源。

7.1.2 Fiddler 与 Shadowsocks 冲突

在安装了 Fiddler 后，浏览器直接通过 Shadowsocks 代理无法科学上网了。估计是流量无法经过 HTTP 代理层到 Socket 代理层。Socks 不要求应用程序遵循特定的操作系统平台，Socks 代理与应用层代理、HTTP 层代理不同，Socks 代理只是简单地传递数据包，而不必关心是何种应用协议（比如 FTP、HTTP 和 NNTP 请求）。

7.2 Web Server

7.2.1 CIDR(Classless Inter-Domain Routing)

CIDR¹(Classless Inter-Domain Routing) 采用各种长度的"网络前缀"来代替分类地址中的网络号和子网号，其格式为：IP 地址 = {<网络前缀>,<主机号>}。为了区分网络前缀，通常采用"斜线记法"（CIDR 记法），即 IP 地址/网络前缀所占比特数。例如：192.168.24.0/22 表示 32 位的地址中，前 22 位为网络前缀，后 10(32-22=10) 位代表主机号。在换算中，192.168.24.0/22 对应的二进制为：1100 0000(192)，1010 1000(168)，0001 1000(24)，0000 0000(0) 其中红色为主机号，总共有 10 位。当这 10 位全为 0 时，取最小地址 192.168.24.0，当这 10 位全为 1 时，取最大地址 192.168.27.255。本例中将第三段地址数据中最小是 00011000 (24)，最大是 00011011 (27)，第四段地址数据中

¹https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing

最小为 0000 0001 (1)，最大为 1111 1110 (254)，以上括号中数据为十进制，其前面为二进制。所以本例中 192.168.24.0/22 对应地址段为 192.168.24.1-192.168.27.254，共 4 个网段。

Table 7.1: IPv4 CIDR

地址格式	网段	子网掩码	主机数
a.b.c.0/24	+0.0.0.255	255.255.255.0	256

7.2.2 maxPostSize

maxPostSize=0 表示 post 请求不限制大小, 从 apache-tomcat-7.0.63 开始, 参数 maxPostSize 的含义改变: 如果将值设置为 0, 表示 POST 最大值为 0, 不限制 POST 大小需将值设置为 -1。在此版本之前设置为 0 表示不限制 POST 大小。The maximum size in bytes of the POST which will be handled by the container FORM URL parameter parsing. The limit can be disabled by setting this attribute to a value less than zero. If not specified, this attribute is set to 2097152 (2 megabytes)². 在 7.0.63 版本之后要设置成小于 0 的数字才表示不受限制, 另外此参数只适用于 request 的 Content-Type 为 “application/x-www-form-urlencoded”。排查此问题, 可以在服务端查看接收到的消息体内容:

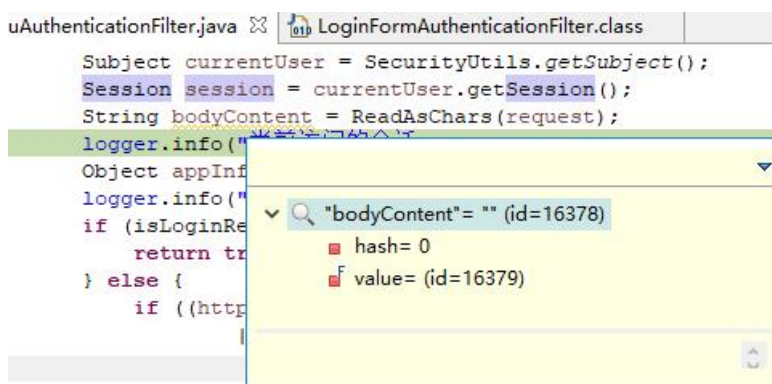


Figure 7.2: 登录接收到的消息体

²<https://tomcat.apache.org/tomcat-8.0-doc/config/http.html>

从图7.2中可以看出，接收到的消息体为空，登录时用户名和密码放在消息体中，不可能为空，那么可以定位到由于服务端没有接收到请求的消息体导致程序问题。

7.2.3 URL 自动重定向

在开启 Fiddler 代理的情况下 (Fiddler 代理端口 8888)，在浏览器中访问 8082 端口时，不管输入什么地址，都会重定向到一个网址。查看 8888 端口占用情况：

```
1 netstat -ano|findstr 8888
```

a(all) 选项表示显示所有连接和侦听端口，n(number) 表示以数字形式显示地址和端口号,o(own) 选项表示显示拥有的与每个连接关联的进程 ID。发现有 3 个进程占用此端口 (Chrome/Fiddler/Nginx)，那么可以肯定是 Nginx 配置了自动跳转：

```
1 server {  
2     listen      8888;  
3     server_name  dolphin.spider.com;  
4  
5     location / {  
6         return 301 https://dolphin.spider.com:443$request_uri;  
7     }  
8 }
```

将 Nginx 监听 8888 端口调整即可解决此问题。

IV

Book

8 Paper 91

- 8.1 Font
- 8.2 纸的种类

Bibliography 93

- Books
- Articles



8. Paper

8.1 Font

8.1.1 类型

8.2 纸的种类

8.2.1 宣纸 (Xuan Paper)

宣纸又名泾县纸，出产于安徽省宣城泾县，以府治宣城为名，故称“宣纸”。宣纸是一种主要供中国毛笔书画以及装裱、拓片、水印等使用的高级艺术用纸张。宣纸拥有良好的润墨性、耐久性、不变形性以及抗虫性能，是中国纸的代表品种。宣纸起于唐代，历代相沿，由于宣纸有易于保存，经久不脆，不会褪色等特点，故有“纸寿千年”之誉¹。

¹<https://zh.wikipedia.org/wiki/%E5%AE%A3%E7%BA%B8>

8.2.2 连史纸

8.2.3 美浓和纸

美浓和纸的历史起源于正仓院珍藏的户籍。据说薪火传承的手抄纸至今已经有约1300年之久的历史。只有纯手工制作才能展现的质感与出色特质至今依然丝毫不变。



Bibliography

Books

Articles

