

# 文章集合

注解

Dolphin



Copyright © 2017 Xiaoqiang Jiang

EDITED BY XIAOQIANG JIANG

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

All Rights Reserved.

*Version 16:32, September 21, 2019*



# Contents

I	风	1
1	2019 年 .....	3
1.1	10 月	3
1.1.1	Spring 请求中 Controller 的定位过程 .....	3
1.1.2	Wireshark Filter .....	3
1.2	血液检测项目	3
1.2.1	项目基本信息 1 .....	3
1.2.2	使用小程序 UI 组件库 .....	4
1.2.3	youtube-dl 使用 .....	4
1.3	09 月	5
1.3.1	MyBatis 分页插件 .....	5

1.3.2	Java 8 集合使用	5
1.3.3	yarn 使用心得	6
1.3.4	Jenkins 部署前端应用	6
1.3.5	搭建私有 Docker 镜像仓库	7
1.3.6	Maven 编译 JavaFX	7
1.3.7	'overlay2' is not supported over nfs	9
1.3.8	Docker 日志	10
1.3.9	MyBatis 启动超时问题	11
1.3.10	Docker Overlay 占满问题处理	11
1.3.11	删除 ElasticSearch 数据	12
1.3.12	MyBatis IOException	14
1.3.13	ElasticSearch OutOfMemoryError: Java heap space	14
1.3.14	磁盘占用分析 (Disk Usage Analysis)	15
1.3.15	开发工具	15
1.3.16	No route to host	16
1.3.17	自动启动应用 (Auto Start App)	17
1.3.18	解决 Skim 开多个窗口问题	19
<b>1.4</b>	<b>08 月</b>	<b>20</b>
1.4.1	JVM 监控	20
1.4.2	can't connect to server Jenkins Publish Over SSH	21
1.4.3	MyBatis 事务	21
1.4.4	MySQL 事务	21
1.4.5	网络数据推送	23
1.4.6	Spring 常见问题	23
1.4.7	Spring Cloud	23
1.4.8	Java 进程突然退出	24
1.4.9	微服务链路追踪 Zipkin	25
1.4.10	Kafka 基础概念	27
1.4.11	Swagger UI 使用总结	30
1.4.12	IntelliJ Idea 项目 *.iml 文件	31
1.4.13	Gradle	31
1.4.14	Spring Bean 初始化顺序	32
1.4.15	Spring 加载完成后执行	34

1.4.16	Apollo 配置的加载时机	35
1.4.17	Jenkins 远程执行 Shell	35
1.4.18	消息队列通用的消息持久化	36
1.4.19	Kibana 使用心得	37
1.4.20	函数式接口之——Predicate	38
1.4.21	Java 8 Stream	38
1.4.22	MySQL 常见操作	39
1.4.23	top 命令	39
1.4.24	Grep 使用小技巧	40
1.4.25	空指针异常	40
1.4.26	Kubernetes 部署	41
1.4.27	创建 kube-controller-manager 集群	69
1.4.28	Tampermonkey 脚本收藏	82
1.4.29	zsh 拷贝文件时通配符解析	82
1.4.30	Git 慢	83
1.4.31	排他锁更新记录	84
1.4.32	MyBatis 代码自动生成	85
1.4.33	参数校验	89
1.4.34	filebeat 递归读取文件	89
1.4.35	Spring 手动获取 Bean	90
1.4.36	Jenkins 利用 Webhook 执行自动构建	91
1.4.37	Jackson 复杂类型反序列化	91
1.4.38	Swagger 使用总结	92
1.4.39	Zuul 统一集成 Swagger2 文档	94
1.4.40	Java 为什么要引入 Lambda 表达式	96
1.4.41	Java 为什么要引入 InvokeDynamic 指令	97
<b>1.5</b>	<b>07 月</b>	<b>97</b>
1.5.1	IntelliJ Idea 闪退	97
1.5.2	Bash 使用 Map 数据结构	98
1.5.3	macOS Mojave 升级 Bash 到 4.0 版本	100
1.5.4	Websocket 常见问题处理	100
1.5.5	Shell 常用脚本	102
1.5.6	容器编排 (Container Orchestration) 方式管理 Docker	102

1.5.7	Docker 方式部署应用	102
1.5.8	EFK Docker 部署	105
1.5.9	集成 Request ID(全链路监测)	106
1.5.10	Feign 传递公共请求头	107
1.5.11	IntelliJ Idea 编译找不到符号	109
1.5.12	微服务常见问题	109
1.5.13	IntelliJ Idea 循环依赖	110
1.5.14	IntelliJ Idea 热部署	111
1.5.15	JVM 内存模型 (Java Memory Model)	112
1.5.16	微服务组件 (Microservice Component)	114
1.5.17	ThreadLocal 共享认证 Token	116
1.5.18	科学上网之 Amazon Cloud 篇 (Fenceless Internet)	117
1.5.19	IntelliJ Idea 常用技巧 (IntelliJ Idea tricks and tips)	119
1.5.20	Mac 快捷键	120
1.5.21	youtube 下载	120
<b>1.6</b>	<b>06 月</b>	<b>121</b>
1.6.1	RabbitMQ Docker 部署	121
1.6.2	Websocket 性能测试	122
1.6.3	用 curl 测试接口	124
1.6.4	扫描如何排除特定类	126
1.6.5	网站启用 https	126
1.6.6	MySQL 事务隔离级别与锁	130
1.6.7	垃圾回收 (Garbage Collection)	130
1.6.8	JVM 运行时数据区 (JVM Run-time Data Areas)	131
1.6.9	JVM 方法区	133
1.6.10	Docker 容器与宿主机通信	133
1.6.11	Docker 跨主机通信方案	134
1.6.12	Git 提交规范	137
1.6.13	Git 常见问题	138
1.6.14	JVM 的垃圾收集器有哪几种	138
1.6.15	控制反转 (IoC)	140
1.6.16	Maven 常见问题解决方案	140
1.6.17	Mac 使用技巧	143





3

Paper

177

3.1

Font

177

3.1.1

类型

177

3.2

纸的种类

177

3.2.1

宣纸 (Xuan Paper)

177

3.2.2

连史纸

178

3.2.3

美浓和纸

178

Bibliography

179

Books

179

Articles

179

---


参考资料<https://zh.wikisource.org/zh/%E8%A9%A9%E7%B6%93>:



风

1	2019 年 .....	3
1.1	10 月	
1.2	血液检测项目	
1.3	09 月	
1.4	08 月	
1.5	07 月	
1.6	06 月	
1.7	05 月	
1.8	03 月	





# 1. 2019 年

## 1.1 10 月

### 1.1.1 Spring 请求中 Controller 的定位过程

### 1.1.2 Wireshark Filter

```
tcp.port == 80 && (websocket) && ip.dst eq 47.101.208.234
```

## 1.2 血液检测项目

### 1.2.1 项目基本信息 1

原型地址<sup>1</sup>。

---

<sup>1</sup>[https://lanhuapp.com/web/#/item/project/product?pid=c177494f-8df2-4715-b265-fc53a0bf015c&docId=fa668098-933a-4103-8f35-5bd2097079b4&docType=axure&image\\_id=fa668098-933a-4103-8f35-5bd2097079b4&pageId=dd56e0302fd64257ac6410675e9af42e&parentId=b368a55f-2f56-408c-b78a-05be8514bf72&type=share\\_mark&tab=product&teamId=3fae9882-dc26-4c11-9976-0a0](https://lanhuapp.com/web/#/item/project/product?pid=c177494f-8df2-4715-b265-fc53a0bf015c&docId=fa668098-933a-4103-8f35-5bd2097079b4&docType=axure&image_id=fa668098-933a-4103-8f35-5bd2097079b4&pageId=dd56e0302fd64257ac6410675e9af42e&parentId=b368a55f-2f56-408c-b78a-05be8514bf72&type=share_mark&tab=product&teamId=3fae9882-dc26-4c11-9976-0a0)

Table 1.1: 基本信息

项	值	备注
项目文档地址	<a href="https://shimo.im/docs/pbhucIuK5MwsRG0E">https://shimo.im/docs/pbhucIuK5MwsRG0E</a>	

1.2.2 使用小程序 UI 组件库

综合对比了下，目前小程序 UI 组件库有

1.2.3 youtube-dl 使用

最近发现了一个 ASMR<sup>2</sup>(Autonomous sensory meridian response，即自发性知觉经络反应) 主播，想要下载她的所有视频，最快的方法是根据她的所有视频的播放列表进行下载，输入如下命令完成：

```
1 youtube-dl -f-i PL110zBd0rRb6aaU-TfH_z5pU-VuDbv
```

下载效果如图所示，-i 参数后直接是播放列表的 ID，-f 参数指定视频的格式，这里是指定的最优质的格式 (best)：

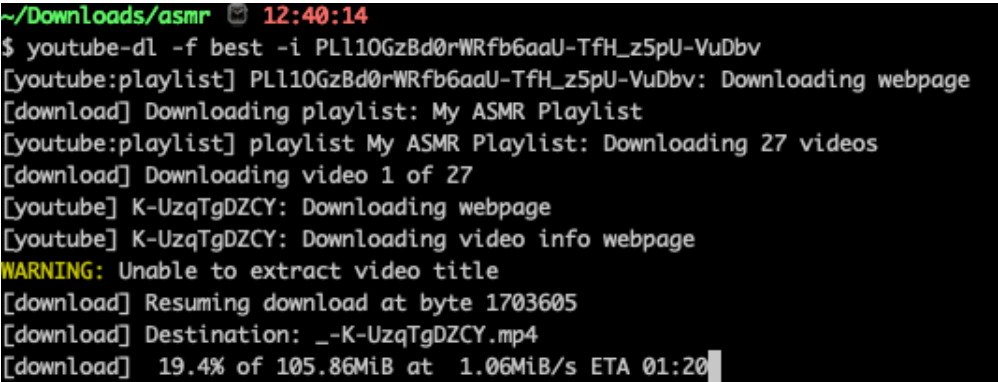


Figure 1.1: youtube-dl 下载播放列表

abb806c0b&param=26f6673c-b7a2-47bb-b229-bc1f256759a2

<sup>2</sup>参考维基百科: [https://en.wikipedia.org/wiki/Autonomous\\_sensory\\_meridian\\_response](https://en.wikipedia.org/wiki/Autonomous_sensory_meridian_response)



## 1.3 09 月

### 1.3.1 MyBatis 分页插件

直接返回 pageInfo 对象：

```
1 PageInfo pageInfo = PageHelper.startPage(query.getPageNum(), query
    .getPageSize()).doSelectPageInfo(new ISelect() {
2     @Override
3     public void doSelect() {
4         pageParams.addAll(customReportProfitMapper.profitPage(
            query));
5     }
6 });
```

### 1.3.2 Java 8 集合使用

根据对象的某个字段进行去重，默认的 distinct 方法无法做到，使用 filter 来实现：

```
1 List<RoomSetting> collect = roomSettings.stream()
2     .filter(distinctByKey(RoomSetting::getName))
3     .collect(Collectors.toList());
4
5 public static <T> Predicate<T> distinctByKey(Function<? super T,
6     Object> keyExtractor) {
7     Map<Object, Boolean> seen = new ConcurrentHashMap<>();
8     return object -> seen.putIfAbsent(keyExtractor.apply(object),
9         Boolean.TRUE) == null;
10 }
```

集合中的对象根据条件进行分组，有一个 List<User>，User 有 name 和 age。现在我们把这个 List 按年龄分成三组：0<age<=20, 20<age<=40, 40<age。

```
1 Map<String, List<User>> tripleUsers = users.stream()
2     .collect(Collectors.groupingBy((Function<User, String>)
```

```
        user -> {  
3      String key;  
4      if (user.getAge() <= 20) {  
5          key = "less20";  
6      } else if (user.getAge() <= 40) {  
7          key = "less40";  
8      } else {  
9          key = "more40";  
10     }  
11     return key;  
12 }, Collectors.toList());
```

### 1.3.3 yarn 使用心得

yarn v1.17.3 运行命令时提示超时错误: There appears to be trouble with your network connection. Retrying。添加超时设置即可:

```
1 yarn install --network-timeout 100000
```

yarn 相对于 npm 的优点目前体会较深的是下载依赖更加快速, 可保证不同平台版本一致性, 但是在实际使用中 Linking Dependencies 这一步非常耗时, 目前已经跑了一个多小时了, What F\*\*K, Stack Overflow 上有简单的解释<sup>3</sup>。有的说是由于 Windows Fefender 导致, 没有任何依据的胡扯。

### 1.3.4 Jenkins 部署前端应用

在 Jenkins 容器中部署 NodeJS 需要安装 NodeJS Plugin 插件, 部署完毕 NodeJS 后, node 的路径在 \$JENKINS\_HOME/tools, 一般 Jenkins 的 home 目录是 /var/jenkins\_home。安装完毕后, npm 没有加到默认路径中, 无法直接使用 npm 命令, 创建 npm 软连接:

---

<sup>3</sup>参考链接: <https://stackoverflow.com/questions/50683248/what-does-linking-dependencies-during-npm-yarn-install-really-do>

```
1 ln -s /var/jenkins_home/tools/jenkins.plugins.nodejs.tools.  
    NodeJSInstallation/nodejs-10.16.3/bin/npm /usr/local/bin/  
    npm
```

### 1.3.5 搭建私有 Docker 镜像仓库

和 Maven 的管理一样，Docker 不仅提供了一个中央仓库，同时也允许我们使用 registry 搭建本地私有仓库。使用私有仓库有许多优点：其一是节省网络带宽，针对于每个镜像不用每个人都去中央仓库上面去下载，只需要从私有仓库中下载即可，其实这不是主要的原因，从中央仓库下载压力大的不是自己，而是中央仓库。关键的点是许多镜像在国内是无法访问的，例如部署 Kubernetes 时，Google 提供的镜像，而要顺利部署，就需要采取各种曲线的方式，是在是不胜其烦。其二是提供镜像资源利用，针对于公司内部使用的镜像，推送到本地的私有仓库中，以供公司内部相关人员使用。

```
1 docker pull registry
```

默认情况下，会将仓库存放于容器的/tmp/registry 目录下，这样如果容器被删除，则存放于容器中的镜像也会丢失，所以我们一般情况下会指定本地一个目录挂载到容器的/tmp/registry 下

```
1 docker run -d -p 5000:5000 -v /data/docker/registry:/tmp/registry  
    registry
```

测试私有镜像：

```
1 docker pull busybox  
2 docker tag busybox 172.19.104.231:5000/busybox  
3 docker push 172.19.104.231:5000/busybox
```

### 1.3.6 Maven 编译 JavaFX

项目中用到了 JavaFX 中的一个类，但是服务端 Docker 的 Jenkins 中没有安装 Java，导致无法编译。Sun 在被 Oracle 收购前的 2008 年年底推出 JavaFX 1.0，希望 Java 在 RIA（Rich Internet Application）方面有所建树。但就从界面来看，非常类似

于 WinForm，如果是被 WinForm 惯坏了的娃，接触 JavaFX 时，会不会有回到上个世纪的感觉。在 WinForm 中，UI 控件刷刷刷直接拖放即可，在 Java 这边图形控件还要哼哧哼哧撸代码，在内部网的企业应用基础的 CRUD 表单操作，借助企业自身开发的建模工具，WinForm 可以做到零编码，这是 Java 图形开发方面很难想象的 (Java 通过定制开发也可以做到)。Oracle 在 2011 年与 JDK8 一起推出了基于原生 Java 重构的 JavaFX2.0，放弃了原先采用 JavaFX Script 机制。在 2014 年，与 JDK8 一起发布了 JavaFX8，从此 JavaFX 成为 JDK 的一部分，从 Java 8 往后，JavaFX 又剥离出了 JDK。Java UI 经历了 AWT、Swing、SWT、JavaFX 几个大的阶段。

### AWT

AWT(Abstract Window ToolKit，抽象窗口工具包)是 SUN 在 1996 年推出的 UI 框架。由于需要跨平台，所以 AWT 只能支持主流平台共有的界面组件和特性(交集)，例如标签、按钮等，这就导致了 AWT 在组件丰富程度以及功能性等方面必然有欠缺。AWT 通过创建一个与操作系统对应的 Peer(对等组件)来实现组件在界面上的显示，依靠操作系统本地方法实现图形功能，属于重量级的 UI 框架。也就是说，当我们使用 AWT 构建图形界面时，实际上是调用操作系统图形库的功能，在不同的操作系统上有不同的表现。

### Swing

Swing 是在 AWT 基础上构建的 UI 框架，全部使用 Java 编写。Swing 提供了比 AWT 更轻量级、更丰富的 UI 组件，如树、列表、表格等。AWT 是基于操作系统本地方法，所以运行速度较快，Swing 是基于 AWT 的 Java 程序，采用解释方法执行。Swing 绘制的界面在不同的操作系统上表现一致。Swing 做得很漂亮、易用性很好的应用有 Eclipse、IntelliJ IDEA，用友 NC6，以及 2BizBox ERP 系统等。还有诸如提供皮肤和 UI 组件的 Substance、SwingX 等第三方开源库。

### SWT

SWT(Standard Widget Toolkit，标准窗口部件)是 IBM 推出的开源 UI 组件库，希望解决 AWT 和 Swing 运用方面的问题。SWT 没有被包含在 JDK 或 JRE 中，不属于 JAVA 原生的 UI 框架。但 SWT 的运用较广泛，如 Eclipse、WebSphere 等安装程序、MAT 等。SWT 提供了比 Swing 更丰富好用的界面组件以及特性，与 AWT 一样，SWT 通过 Peer 调用操作系统本地方法。同时，SWT 通过使用特定操作系统的特性，加快 UI 组件响应时间，所以 SWT 需要为不同的操作系统准备安装包。与 AWT 一样，没

有 Swing 的 LookAndFeel 功能<sup>4</sup>。

在 Maven(版本 3.6.1) 中编译 JavaFX 首先下载 JDK(版本 1.8.0\_211)。在 pom 中指定 fxrt.jar 路径即可<sup>5</sup>。

```

1 <dependency>
2   <groupId>javafx</groupId>
3   <artifactId>jfxrt</artifactId>
4   <version>${java.version}</version>
5   <scope>system</scope>
6   <systemPath>${java.home}/jre/lib/ext/jfxrt.jar</systemPath>
7 </dependency>

```

### 1.3.7 'overlay2' is not supported over nfs

Docker 启动时提示: 'overlay2' is not supported over nfs, Error starting daemon: error initializing graphdriver: backing file system is unsupported for this graph driver。大意是 overlay2(Linux kernel 4.0 以后才支持) 文件系统无法工作于网络文件系统 (Net File System) 之上。Overlay 第一个版本是使用 3.18 以上的内核, 由于一些功能的欠缺, 使用了很多 hard link, 从而可能导致 inode 过多的问题。在 4.0 内核以上, 解决了这个问题, 所以在 Docker 1.12 有了 Overlay2 以解决这个问题。原来阿里云主机的数据盘是通过网络文件系统的方式挂载, 如图1.2所示。

```

[root@i7n663-6m9g1d0jk01t9Z docker]# df -Th /data
Filesystem                                Type      Size  Used Avail Use% Mounted on
375551818-1.cn-shanghai.nas.aliyuncs.com:/ nfs4      10P   227G   10P    1% /data

```

Figure 1.2: 阿里云挂载 nfs 数据盘

而 Docker 在一般情况下默认不支持, 将配置文件/etc/docker/daemon.json 路径调整为本地路径即可解决问题:

```

1 {

```

<sup>4</sup>以上内容抄袭自: <https://zhuanlan.zhihu.com/p/30694468>

<sup>5</sup>参考自: <https://stackoverflow.com/questions/15278215/maven-project-with-javafx-with-jar-file-in-lib>

```
2 "data-root": "/var/lib/docker",  
3 "storage-driver": "overlay2"  
4 }
```

注意使用 overlay2 文件系统时，默认文件夹下会创建 overlay2 文件夹，不会有 devicemapper 文件夹。选择 devicemapper 文件系统时，会创建一个 100GB 大小的文件，但是不回实际占用掉 100GB 的空间。

### 1.3.8 Docker 日志

假设 application 是 Docker 容器内部运行的应用，那么对于应用的第一部分标准输出 (stdout) 日志，Docker Daemon 在运行这个容器时就会创建一个协程 (goroutine)，负责标准输出日志。由于此 goroutine 绑定了整个容器内所有进程的标准输出文件描述符，因此容器内应用的所有标准输出日志，都会被 goroutine 接收。goroutine 接收到容器的标准输出内容时，立即将这部分内容，写入与此容器一对应的日志文件中，日志文件位于 /var/lib/docker/containers/<container\_id>，文件名为 <container\_id>-json.log。至此，关于容器内应用的所有标准输出日志信息，已经全部被 Docker Daemon 接管，并重定向到与容器一对应的日志文件中。常用日志命令如下：

```
1 # 查看 xxl-job 容器的输出日志  
2 # 4bf303105abb 是 xxl-job 容器 ID 的缩写  
3 docker logs 4bf303105abb
```

注意此命令查看 Docker 日志默认输出是从日志的最开始输出 (个人认为此处默认显示最新日志较为妥当，开始时以为默认输出最新的内容，类似于 tail 命令的输出，每次查看日志时都有大量错误信息，后来才发现是容器在几个月之前开始运行时产生的错误，看到的日志输出已经是几个月之前的内容)，一般情况下的习惯是，如果日志文件较大，不太可能从日志文件开始检查，一般只需要查看最新的日志即可，如果希望 Docker logs 查看最新的输出日志，使用如下命令：

```
1 # 查看 xxl-job 容器的最新输出日志  
2 # t 参数会输出详细的时间戳，便于查看分析  
3 docker logs -t --tail 300 4bf303105abb
```

查看日志文件已经超过了 6GB，使用如下命令删除掉旧日志。

```
1 # 保留日志最新 300 行内容
2 tail -n 300 big-file-json.log >> new.log
3 # 删除旧日志
4 mv new.log big-file-json.log
```

### 1.3.9 MyBatis 启动超时问题

在启动 MyBatis 项目时提示错误：

```
1 org.springframework.beans.factory.BeanDefinitionStoreException:
   IOException parsing XML document from class path resource
   [mybatis/mybatis-config.xml]; nested exception is java.net
   .ConnectException: Operation timed out (Connection timed
   out)
```

原因是项目的 mybatis-config.xml 文件中的文档类型定义 DTD(Document Type Define) 地址无法访问导致，调整地址即可解决此问题，2 个地址，哪个可以访问使用哪个(官方使用的是未加 www 的地址)，最后发现是 WIFI 网络问题，也许网络有安全策略。

```
1 http://mybatis.org/dtd/mybatis-3-config.dtd
2 http://www.mybatis.org/dtd/mybatis-3-config.dtd
```

### 1.3.10 Docker Overlay 占满问题处理

Apollo 客户端获取配置时提示：

```
1 Long polling failed, will retry in 32 seconds. appId: 0010020009,
   cluster: default, namespaces: application+TEST1.EUREKA+
   TEST1.DATASOURCE-DRUID+TEST1.REDIS-CONFIG, long polling
   url: null, reason: Get config services failed from http
   ://127.0.0.1:1808/services/config?appId=0010020009&ip
   =192.168.2.80
```

直接获取网页时返回空，正常情况应该返回应用的基础配置信息。Apollo 容器突然宕掉，登陆容器查看/opt/logs/100003171/apollo-assembly.log 日志，原来是云服务器磁盘满了，目前磁盘空间在 40GB。overlayfs 集成进了 linux 3.18 内核。overlay 存储驱动主要使用的是 overlayfs 技术。中文名是叠合式文件系统。多个文件系统可以 mount 之后进行合并。OverlayFS 是一个类似于 AUFS 的现代联合文件系统，但更快，实现更简单。Docker 为 OverlayFS 提供了一个存储驱动程序。OverlayFS 是内核提供的文件系统，overlay 和 overlay2 是 docker 提供的存储驱动。OverlayFS 将单个 Linux 主机上的两个目录合并成一个目录。这些目录被称为层，统一过程被称为联合挂载。OverlayFS 关联的底层目录称为 lowerdir，对应的高层目录称为 upperdir。合并过后统一视图称为 merged。查看 Docker 磁盘占用情况：

```
1 docker system df
```

要解决根本问题，需要将 Docker 容器的目录迁移到磁盘空间较大的目录，目前挂载有一个 500GB 的目录。迁移步骤如下：

```
1 # 停止 Docker 服务
2 systemctl stop docker
3 # 迁移文件
4 rsync -avz /var/lib/docker /home/docker/lib/
```

### 1.3.11 删除 Elasticsearch 数据

随着系统的运行 Elasticsearch 中的数据不断增加，目前已经有 180GB 以上了，磁盘空间告警，目前是测试环境，旧的日志数据少有分析价值，果断删除。在 Elasticsearch 主机上使用如下命令删除文档：

```
1 curl -H'Content-Type:application/json' -d'{
2     "query": {
3         "range": {
4             "@timestamp": {
5                 "lt": "now-7d",
6                 "format": "epoch_millis"
7             }
8         }
9     }
10 }
```



```
8     }
9   }
10 }
11 ' -XPOST "http://127.0.0.1:9200/*-*/_delete_by_query?
    wait_for_completion=false&pretty"
```

注意删除文档不会立即将文档从磁盘删除，只是将文档标记为已删除状态<sup>6</sup>。随着不断的索引更多的数据，Elasticsearch 将会在后台清理标记为已删除的文档。`wait_for_completion=false` 表示不需要等待执行结果，即异步执行，执行命令后，将返回一个任务的 ID 编号，通过 ID 编号，可以查看和取消任务。执行了删除动作之后，查看当前 ElasticSearch 存储的数据并不会立即释放，要执行 `merge` 操作之后才会释放：

```
1 # 查看文档大小是否释放
2 duc ls -Fg /data/docker/efk/elasticsearch-7.2.0/data/nodes/0/
3 # 执行 merge 操作，释放空间
4 curl -H'Content-Type:application/json' -XPOST 127.0.0.1:9200/*/_forcemerge?max_num_segments=1
5 # 查看当前的 task
6 curl -H'Content-Type:application/json' -XGET 127.0.0.1:9200/_tasks
```

执行 `delete` 操作时，最大的问题是速度非常慢，执行 `merge` 操作后，才会真正释放磁盘空间，所以不推荐采用此种方式来删除文档，释放磁盘空间。而自动化 `merge` 触发是随着段数越来越多，这些段会定期合并为更大的段。这一过程称为合并。由于所有段 (Segment) 都是不可更改的，这意味着在索引期间所用磁盘空间通常会上下浮动，这是因为只有合并后的新段创建完毕之后，它们所替换的那些段才能删掉。合并是一项极其耗费资源的任务，尤其耗费磁盘 I/O<sup>7</sup>。更好的方式是直接删除索引 (Index)，首先查看当前 ElasticSearch 包含哪些索引：

```
1 # 查看节点列表
2 curl 'localhost:9200/_cat/nodes?v'
3 # 查看节点索引
```

<sup>6</sup>来源: <https://www.elastic.co/cn/blog/lucenes-handling-of-deleted-documents>

<sup>7</sup>来源: <https://www.elastic.co/cn/blog/how-many-shards-should-i-have-in-my-elasticsearch-cluster>

```
4 curl -XGET -H'Content-Type:application/json' 127.0.0.1:9200/_cat/  
    indices
```

索引里有 `store.size` 和 `pri.store.size` 字段, `store.size` 表示 primary 和 replica shards 占用的空间大小, `pri.store.size` 仅仅表示 primary shards 空间大小。Elasticsearch 提供了将索引划分成多份的能力, 这些份就叫做分片 (Shard)。每个分片本身也是一个功能完善并且独立的“索引”, 这个“索引”可以被放置到集群中的任何节点上。允许你在分片 (潜在地, 位于多个节点上) 之上进行分布式的、并行的操作, 进而提高性能/吞吐量。删除索引:

```
1 # 删除索引  
2 curl -XDELETE -H'Content-Type:application/json' 'localhost:9200/  
    filebeat-7.2.0-2019.08.29-000004?pretty'
```

相对于删除 Elasticsearch 文档 (Document) 需要一个多小时, 删除索引 (Index) 几秒内即可完成。除了在后台通过命令删除文档, 也可以在 Kibana 界面上 Management->ElasticSearch->Index Management 维护索引。

### 1.3.12 MyBatis IOException

在应用启动时, 提示 `org.springframework.beans.factory.BeanDefinitionStoreException: IOException parsing XML document from class path resource [mybatis/mybatis-config.xml]; nested exception is java.net.ConnectException: Operation timed out (Connection timed out)`, 连接超时, 开启系统全局代理问题解决。

### 1.3.13 Elasticsearch OutOfMemoryError: Java heap space

ES 的 data node 存储数据并非只是耗费磁盘空间的, 为了加速数据的访问, 每个 segment 都有会一些索引数据驻留在 heap 里。因此 segment 越多, 瓜分掉的 heap 也越多, 并且这部分 heap 是无法被 GC 掉的! 理解这点对于监控和管理集群容量很重要, 当一个 node 的 segment memory 占用过多的时候, 就需要考虑删除、归档数据, 或者扩容了。增加 Java Heap 大小启动<sup>8</sup>:

```
1 export ES_JAVA_OPTS="-Xms2g -Xmx2g"
```

<sup>8</sup>来源: <https://www.elastic.co/guide/en/elasticsearch/reference/current/heap-size.html>

```
2 ./elasticsearch
```

### 1.3.14 磁盘占用分析 (Disk Usage Analysis)

一般情况下找到磁盘中的大文件，进而了解磁盘占用情况可以采用命令：

```
1 find / -type f -size +800M -print0 | xargs -0 du -h | sort -nr
```

此命令会找到磁盘中的大文件，移除部分不再使用的大文件即可释放磁盘空间，缺点是如果磁盘中文件较多时，效率往往很低下，很久无法看到结果，ncdu 工具来分析也有类似的缺点。duc(似乎可以把这个工具理解为 Disk Usage Cache 的含义) 工具具备缓存扫描结果的特性，Duc is a collection of tools for indexing, inspecting and visualizing disk usage. Duc maintains a database of accumulated sizes of directories of the file system, and allows you to query this database with some tools, or create fancy graphs showing you where your bytes are<sup>9</sup>. 还可以以图形化的方式分析磁盘文件占用情况。

```
1 # 索引根目录下的 data 文件夹
2 duc index /data
3 # 列出目录大小
4 duc ls -Fg /data/docker/efk
```

运行索引的时间较长，查看文件大小就是从索引结果获取，瞬间即可获取到磁盘文件占用情况。

### 1.3.15 开发工具

开发过程中常用到的工具，GitLab 用作原始码管理平台，Jenkins 作为原始码提交后的自动 CI 工具，Jira 作为 Bug 与项目管理工具，缺点是商业产品，价格不菲，RabbitMQ 作为消息队列工具。Zipkin 作为微服务的性能管理工具 (Application Performance Management)，缺点还是对应用代码有一定侵入性，后面考虑使用 Service Mesh。数据库使用 MySQL，缺点是现在 MySQL 被商业公司 Oracle 收购，未来发展有许多不确定性，可以考虑尝试 BSD 协议的 PostgreSQL。

<sup>9</sup>来自官方网站<https://duc.zevv.nl/>的描述。

Table 1.2: 开发工具集合

主机	名称	位置	用途
APP	Redis		缓存
APP	Apollo		应用配置管理工具
APP	Filebeat	/opt/org/tools	轻量级日志搜集工具
CI	GitLab		源码管理
CI	Jira		项目管理 & Bug 管理
CI	RabbitMQ		消息队列
CI	Nexus		依赖缓存仓库管理工具
CI	Jenkins		持续集成工具
EFK	Kibana(5601)	/data/docker/efk	日志管理工具
EFK	Zipkin		分布式 APM
EFK	ElasticSearch(9200)	/data/docker/efk	全文索引文件储存
EFK	MySQL		数据库

1.3.16 No route to host

在 xxl-job 中访问执行器时，抛出 NoRouteToHostException 异常，在 Docker 容器中访问执行器端口同样抛出 No route to host 异常，从其他局域网主机访问执行器地址同样抛出 No route to host 异常。查看主机 172.19.150.82，防火墙服务和 iptables 服务皆已经关闭。按照逻辑，如果关闭防火墙后，所有端口应该是处于开放状态。但是此时却无法访问端口，经过检查，可能是当前防火墙关闭只针对当前会话有效，对其他会话无效。执行如下操作开启端口：

```
1 # 启动防火墙
2 systemctl start firewalld
3 # 开启端口
4 firewall-cmd --zone=public --add-port=9997/tcp --permanent
5 # 刷新 firewalld
6 firewall-cmd --reload
```

再次访问服务即可。如果防火墙拒绝了流量，使用 nc 命令一般提示 No route to

host, 如果防火墙开启, 但是服务端没有进程监听端口, 一般提示连接拒绝 (Connection refused)。

```
1 # 服务端防火墙未开启 9995 端口
2 [root@iZuf63refzwegld9dh94t9Z ~]# nc -v -z -w2 172.19.150.82 9995
3 Ncat: Version 7.50 ( https://nmap.org/ncat )
4 Ncat: No route to host.
5 # 服务端端口 9997 允许流量
6 # 但没有进程监听
7 [root@iZuf63refzwegld9dh94t9Z ~]# nc -v -z -w2 172.19.150.82 9997
8 Ncat: Version 7.50 ( https://nmap.org/ncat )
9 Ncat: Connection refused.
```

netcat (通常缩写为 nc) 是一个计算机网络实用程序, 用于使用 TCP 或 UDP 读取和写入网络连接。该命令旨在成为可靠的后端, 可以直接使用或由其他程序和脚本轻松驱动。同时, 它是一个功能丰富的网络调试和调查工具, 因为它可以产生用户可能需要的几乎任何类型的连接, 并具有许多内置功能。它的功能列表包括端口扫描, 传输文件和端口监听, 它可以用作后门<sup>10</sup>。

### 1.3.17 自动启动应用 (Auto Start App)

应用经常有自动退出的情况, 在还没有找到问题之前, 暂时采用自动监视并自动启动的方式。编写如下脚本:

```
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 # 也可以使用命令 set -o nounset
5 set -u
6
7 # 当任何一行命令执行失败时, 自动退出脚本
8 # 也可以使用命令 set -o errexit
9 set -e
```

<sup>10</sup>来源: <https://en.wikipedia.org/wiki/Netcat>

```
10
11 set -x
12
13 JAVA_HOME="/opt/dabai/tools/jdk1.8.0_211"
14 APP_HOME="/data/jenkins/soa-robot-service"
15 APP_NAME="soa-robot-service-1.0.0-SNAPSHOT.jar"
16
17 count=`ps -ef | grep ${APP_NAME} | grep -v "grep" | wc -l`
18 if [[ ${count} -lt 1 ]]; then
19     cd ${APP_HOME}
20     echo "Process terminal by OS" >> ${APP_HOME}/autostart.log
21     nohup ${JAVA_HOME}/bin/java -Xmx440M -Xms128M -jar \
22     -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/opt \
23     -Xdebug \
24     -Xrunjdwp:transport=dt_socket,suspend=n,server=y,address=5009 \
25     ${APP_HOME}/${APP_NAME} >> /dev/null &
26     sleep 5
27 else
28     echo "process already exists!"
29 fi
```

在系统的 `cron` 服务中新建定时任务，定时服务调用脚本监视程序是否在运行，如果检测到应用没有运行，则定时脚本自动启动应用，如下配置每隔 1 分钟执行一次 `autostart.sh` 脚本：

```
1 * * * * root ( sleep 10; /data/jenkins/soa-robot-service/
    autostart.sh )
```

需要注意的是 `crontab`(`cron table`) 目前最小粒度是分钟，分钟以下的触发间隔需要特殊的写法。通常的写法是同时定义多个定时任务，每个定时任务暂停一定的时间，时间互相错开，例如每隔 30 秒执行时，定义 2 个定时任务，第一个任务与第二个任务互相错开 30 秒触发即可。`cron` 名称的由来是希腊语中的时间，也叫 (`chronos`)，一

般译作柯罗诺斯<sup>11</sup>。柯罗诺斯（古希腊语：；英语：Chronos）是古希腊神话中的一位原始神，代表着时间。柯罗诺斯是无实体的神祇，又或者以蛇的形象出现——拥有三颗头：人头、牛头及狮子头。他与配偶阿南刻围着原始世界卵盘绕，然后将之分开，形成了大地、海与天空的有序宇宙。在希腊罗马的马赛克艺术作品中，他被描述成一个转动黄道带的神，名字是“伊恩”（Aeon，意为“永恒”）。而在近代艺术作品中，他通常以手执镰刀的老人形象出现<sup>12</sup>。柯罗诺斯严格来说是非人格化的神，祂就是时间本身，祂的形象也并非人型，而是生有狮头牛首中有一神脸的有翼巨龙/蛇形象，别名赫拉克勒斯。蛇作为时间的象征，可能来源于古人认为蛇不断蜕皮，冬眠时如同僵死而春天又苏醒，是永生不死的，以及认为时间是循环的，而蛇（尤其是衔尾蛇）的形象可以代表循环。也就明白了埃及许多形象都类似蛇，记得法老的手杖就是蛇的造型。秘教传统里宙斯用来缚住瑞亚与其交合的赫拉克勒斯之结，可能也来源于此。时间是第一因（或说第一因不可言说），时间是孕生的力量，所以柯罗诺斯独自生下埃忒尔（以太）、卡俄斯（混沌）、厄瑞玻斯（幽冥），这些都是非人格化的神，又在埃忒尔中产下一个蛋，后钻入蛋中（或说是赫拉克勒斯扼开，但是赫拉克勒斯就是柯罗诺斯），蛋中生出初代神王双性神法涅斯，同时蛋上下分开，上为天，下为地。或说，法涅斯身上缠绕着一条蛇，那蛇便是柯罗诺斯。至此以后，柯罗诺斯不再单独出现。时间能使天地分开，亦能重新使天地归一，时间是开始，也是结束。或者说，时间既无开始，亦永远不曾结束。时间是独一无二的真实的见证，是万物之父，是不可孕生的孕生力量。时间从来就有，是岁月和生命的父。时间永不倦，它在不间断的演化里永恒行进，并自发生成。查看 `cron` 定时任务的执行情况可以使用如下命令：

```
tail -f /var/log/cron
```

应用所有的模拟对象会在一段时间后完全退出，在未找到问题解决问题之前，采用如下方式定时重启：

```
* */6 * * * root /data/jenkins/soa-robot-service/upgrade.sh
```

### 1.3.18 解决 Skim 开多个窗口问题

在使用 TexStudio 编译预览  $\text{\LaTeX}$  文档时，每次编译预览文档 Skim 都会打开一个新的独立窗口，多次编译后打开的应用几乎全部是 Skim，而期望到行为是编译文档

<sup>11</sup>来源：<https://en.wikipedia.org/wiki/Cron>

<sup>12</sup>来源：<https://zh.wikipedia.org/wiki/>

时 Skim 直接刷新当前文档，加载最新的内容即可。解决此问题可打开 Skim 的 Sync 特性，自动监测文件变化并重新加载 pdf 文档，打开 Skim 的 Sync 特性设置在菜单 Skim->Preference->Sync 下，如图1.3所示：

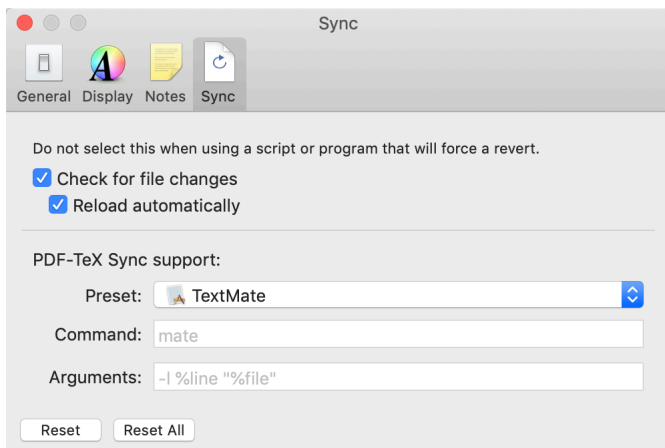


Figure 1.3: Skim 监测文件变化设置

打开此选项后，Skim 会自动监测 pdf 文件的变化并重新加载 (Reload) 文件，但是此时需要注意在 TeXstudio 中执行编译时，不需要再执行编译预览 (Build&View) 命令，直接执行编译 (Build) 动作即可，Skim 会监测到本地 pdf 变化并自动刷新，否则还是会同时打开许多 Skim 窗口，因为预览命令会调用 Skim 打开 pdf 文件命令 (前提是设置了 TeXstudio 的 External PDF Viewer 为 Skim)。

## 1.4 08 月

### 1.4.1 JVM 监控

启动 jstatd:

```
1 ./jstatd -J-Djava.security.policy=jstatd.all.policy
2 /opt/dabai/tools/jdk1.8.0_211/bin/jstatd -J-Djava.rmi.server.
    hostname=172.19.150.82 \
3 -J-Djava.security.policy=/opt/dabai/tools/jdk1.8.0_211/bin/jstatd.
    all.policy \
```



```
4 -J-Djava.rmi.server.logCalls=true
```

### 1.4.2 can't connect to server Jenkins Publish Over SSH

Jenkins 配置 Publish Over SS 时提示错误：can't connect to server。重新刷新页面后生效，多个 Domain 同样的用户名称时不容易区分，如图1.4所示。还有可能是服务器的密码填写错误，有一次出现此问题是由于服务器修改了密码而 Jenkins 中的认证凭据还没有更新导致。

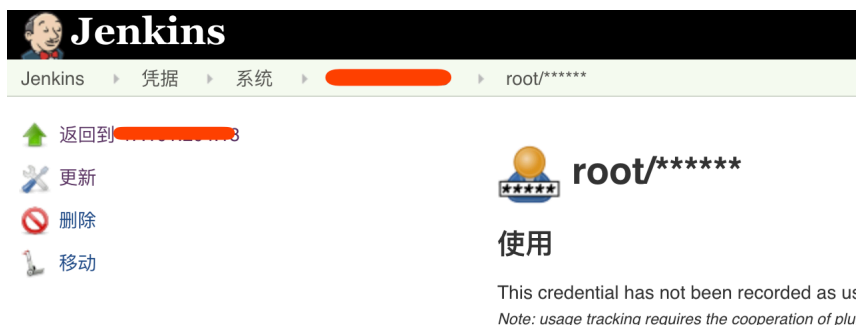


Figure 1.4: Publish Over SSH 凭据配置

### 1.4.3 MyBatis 事务

在 MyBatis 中使用事务时，在方法上添加 Transactional 注解即可。需要注意的是 Transactional 只有来自外部的方法调用才会被 AOP 代理捕获，类内部方法调用本类内部的其他方法并不会引起事务行为，如图1.5所示，一个 Transactional 注解的方法，2 次数据库交互创建了 2 个 SqlSession 会话。

而从类外部正常调用时，打开 MyBatis 的调试输出可以发现，添加注解后整个方法与数据库的交互都是使用的同一个会话 DefaultSqlSession@526e6370，调用 SqlSession 的 getMapper 方法获取 Mapper 和使用注入获取 Mapper 等价，都能够触发事务，在添加 Transactional 注解后的方法内都是使用同一个会话。

### 1.4.4 MySQL 事务

在执行 MySQL 语句时，提示错误：Statement cancelled due to timeout or client request。一般说是由于 MyBatis 超时时间设置不足，在 mybatis-config.xml 文件中调整

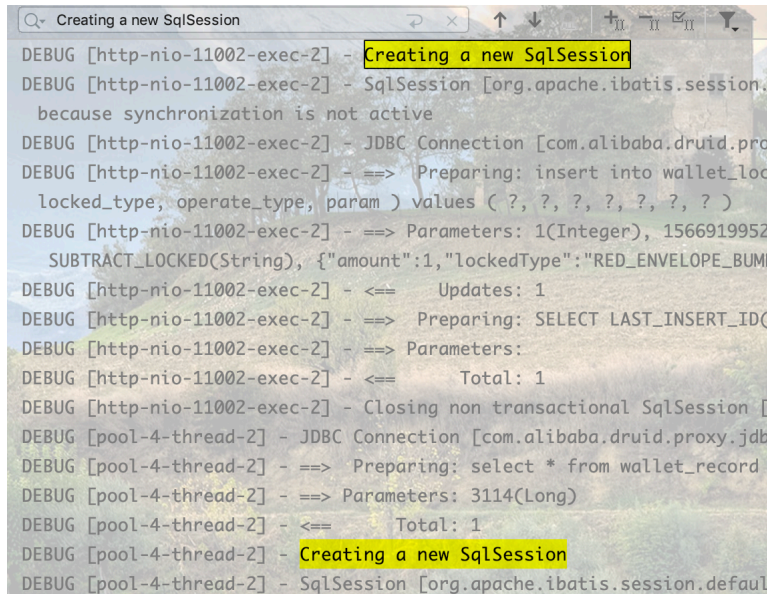


Figure 1.5: 事务未触发时数据库会话

超时时间:

```
1 <setting name="defaultStatementTimeout" value="60"/>
```

Lock wait timeout exceeded; try restarting transaction。最后发现是由于有未提交的事务导致此问题。解决问题采用如下步骤:

```

1 -- 查看当前正在进行的事务
2 select trx_id,trx_state, trx_started, trx_mysql_thread_id,
   trx_query
3 from information_schema.innodb_trx;
4 # 结束事务 (不推荐)
5 # 1336为trx_mysql_thread_id字段
6 kill 1336

```

### 1.4.5 网络数据推送

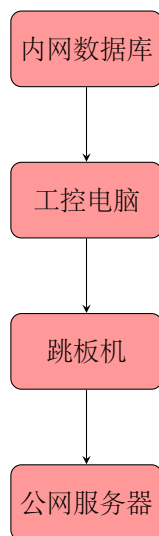


Figure 1.6: 内网数据与公网服务器通信

跳板机与公网服务器可以尝试建立局域网 (WireGuard)，跳板机器可以做端口转发 (Port Forward)，端口转发后，可以直接从公网服务器访问内网的数据库服务。

### 1.4.6 Spring 常见问题

**cannot be delegated to target bean. Switch its visibility to package or protected.**

Spring 启动时提示: Initialization of bean failed; nested exception is IllegalStateException: Need to invoke method 'cacheSync' found on proxy for target class 'ScheduleTask' but cannot be delegated to target bean. Switch its visibility to package or protected. 大意是初始化 Bean 异常，将可见性调整为 protected。将定时任务方法 cacheSync 的可见性调整为 protected 问题解决。问题虽然得到解决，产生疑问的是项目一直未抛出此异常，在引入了新的依赖时出现了此问题，其中的深层原理没有发掘。

### 1.4.7 Spring Cloud

官方说明 Spring Cloud 这些版本号的单词来自于英国伦敦的地铁站站名 (London Tube stations)。当一个版本的 Spring Cloud 项目的发布内容积累到临界点或者一个严重 bug 解决可用后，就会发布一个“service releases”版本，简称 SRX 版本，其中 X 是

一个递增数字<sup>13</sup>。

Table 1.3: Spring Cloud 版本与 Spring Boot 版本兼容关系

组件名称	Spring Boot 版本兼容
Angel	兼容 Spring Boot 1.2.x
Brixton	兼容 Spring Boot 1.3.x, 也兼容 Spring Boot 1.4.x
Camden	兼容 Spring Boot 1.4.x, 也兼容 Spring Boot 1.5.x
Dalston	兼容 Spring Boot 1.5.x
Edgware	兼容 Spring Boot 1.5.x
Finchley	兼容 Spring Boot 2.0.x
Greenwich	兼容 Spring Boot 2.1.x
Hoxton	兼容 Spring Boot 2.2.x

Spring Cloud 微服务启动时显示错误 `SpringApplicationBuilder.<init>([Ljava/lang/Object;)V`。经过排查, 推测是由于 Spring Cloud 版本不兼容问题导致, Spring Cloud 与组件之间版本对应情况可参考这里<sup>14</sup>, 不过暂时没有重现此问题。Greenwich 地铁站属于 Jubilee 线路, 于 1999 年 5 月 14 日开通<sup>15</sup>。

1.4.8 Java 进程突然退出

遇到 Java 进程突然退出, 也没有错误日志和 dump 日志。输入如下命令查看:

```
1 sudo dmesg -T | grep "(java)"
2 # dump内存
3 jmap -dump:live,format=b,file=heap-dump-3.bin 30607
```

可以看出是由于内存溢出 Linux 自动结束掉进程导致此问题。

```
1 [Tue Aug 20 23:59:29 2019] Out of memory: Kill process 16846 (java
   ) score 62 or sacrifice child
```

<sup>13</sup>来源链接: <https://spring.io/projects/spring-cloud>  
<sup>14</sup>来源链接: <https://github.com/spring-projects/spring-cloud/wiki/Spring-Cloud-Greenwich-Release-Notes>  
<sup>15</sup>资料来源: [https://en.wikipedia.org/wiki/List\\_of\\_London\\_Underground\\_stations](https://en.wikipedia.org/wiki/List_of_London_Underground_stations)

```
2 [Tue Aug 20 23:59:29 2019] Killed process 16846 (java) total-  
    vm:20446088kB, anon-rss:1006464kB, file-rss:0kB, shmem-  
    rss:0kB
```

### 1.4.9 微服务链路追踪 Zipkin

分布式实时数据追踪系统（Distributed Tracking System）的解决方案有很多，比如 Google 的 Dapper，微博的 fiery<sup>16</sup>，京东的 Hydra<sup>17</sup>，淘宝的鹰眼，Twitter 的 ZipKin，eBay 的 Centralized Activity Logging (CAL)，大众点评网的 CAT，Uber 的 Jaeger<sup>18</sup>，部分解决方案有开源，当前主流的开源框架是 Twitter 的 zipkin 及 Uber 的 jaeger。Zipkin 是一套 Twitter 开发的开源的 APM 工具 (Application Performance Management)，可以用于微服务的调用链监控。截至 2019 年 8 月 22 日，Zipkin 的总体流程如下：

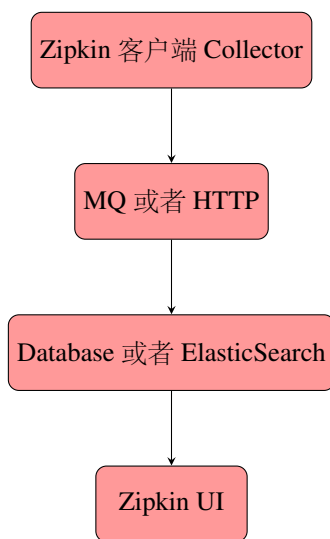


Figure 1.7: Zipkin 数据流转过程

输入如下命令部署 Zipkin 服务端：

<sup>16</sup>Fiery 是一款为 PHP 提供服务的性能跟踪监控系统 (APM-Application Performance Management)，可以方便的查看线上调用关系，响应性能，回放每次请求的具体执行过程、参数、异常。

<sup>17</sup>搜索了半天没有找到，知道有这么个玩意儿就可以了。

<sup>18</sup>项目主页：<https://github.com/jaegertracing/jaeger>

```
1 docker run -d --name zipkin-server \  
2     -p 9411:9411 \  
3     -e "KAFKA_BOOTSTRAP_SERVERS=172.19.150.82:9092" \  
4     -e "STORAGE_TYPE=elasticsearch" \  
5     -e "ES_HOSTS=172.19.150.82:9200" \  
6     -e "ES_INDEX=zipkin" \  
7     -e "ES_INDEX_SHARDS=1" \  
8     -e "ES_INDEX_REPLICAS=1" \  
9     openzipkin/zipkin:2.15
```

在启动容器后,如果需要调整容器参数,可直接到容器配置文件存储目录 (/var/lib/docker/containers/容器 ID) 下进行调整。指定数据存放在 Elasticsearch 中,缓冲消息队列采用的 Kafka。下一步就是要将调用链路相关的数据发送到 Elasticsearch 中,作为 Zipkin 显示的数据来源。微服务中引入相关依赖包:

```
1 <dependency>  
2     <groupId>org.springframework.kafka</groupId>  
3     <artifactId>spring-kafka</artifactId>  
4     <version>2.2.4.RELEASE</version>  
5 </dependency>  
6  
7 <dependency>  
8     <groupId>org.springframework.cloud</groupId>  
9     <artifactId>spring-cloud-starter-sleuth</artifactId>  
10 </dependency>  
11  
12 <dependency>  
13     <groupId>org.springframework.cloud</groupId>  
14     <artifactId>spring-cloud-starter-zipkin</artifactId>  
15 </dependency>
```

新增客户端配置:

```
1 # Zipkin缓冲kafka地址
2 spring.kafka.bootstrap-servers = 172.19.150.82:9092
3 # 设置采样频率，默认为0.1，设置为全采样，便于观测，实际项目中根据具体
   情况设置
4 spring.sleuth.sampler.percentage = 1
5 spring.sleuth.web.client.enable = true
6 spring.zipkin.base-url = http://example.com
7 # 配置zipkin发送类型为kafka
8 spring.zipkin.sender.type = kafka
9 # Kafka主题，不配置时默认使用zipkin
10 spring.zipkin.kafka.topic = zipkin
```

客户端配置新增完毕后，在安装 Kafka 的服务器上，采用如下命令查看微服务跟踪信息是否已经写入到 Kafka 中：

```
1 /data/docker/efk/kafka_2.12-2.3.0/bin/kafka-console-consumer.sh --
   bootstrap-server 172.19.150.82:9092 --topic zipkin
```

使用如下命令查看 Elasticsearch 的所有索引，确认数据是否已经写入到 Elasticsearch 中：

```
1 curl -XGET 127.0.0.1:9200/_cat/indices
```

如果已经写入，会有类似于 zipkin-span-2019-08-20 名字的索引项输出。Brave 是用来装备 Java 程序的类库，提供了面向标准 Servlet、Spring MVC、Http Client、JAX RS、Jersey、Resteasy 和 MySQL 等接口的装备能力，可以通过编写简单的配置和代码，让基于这些框架构建的应用可以向 Zipkin 报告数据。同时 Brave 也提供了非常简单且标准化的接口，在以上封装无法满足要求的时候可以方便扩展与定制。

### 1.4.10 Kafka 基础概念

#### Rebalance

在某些条件下，partition 要在 consumer 中重新分配。以下条件，会触发 rebalance：有新的 consumer 加入，旧的 consumer 挂了，coordinator 挂了，集群选举出新的 coordinator，

topic 的 partition 新加, consumer 调用 `unsubscribe()`, 取消 topic 的订阅。当 consumers 检测到要 rebalance 时, 所有 consumer 都会重走上面的流程, 进行步骤 `JoinGroup + SyncGroup`。当一个 consumer 挂了, 或者有新的 consumer 加入, 其他 consumers 通过 heartbeat 知道要进行 rebalance。

### 分区 (Partition)

`ProducerRecord` 对象包含了目标主题、键和值。Kafka 的消息是一个个键值对, `ProducerRecord` 对象可以只包含目标主题和值, 键可以设置为默认的 `null`, 不过大多数应用程序会用到键。键有两个用途: 可以作为消息的附加信息, 也可以用来决定消息该被写到主题的哪个分区。拥有相同键的消息将被写到同一个分区。

### Producer

创建生产者时提示 `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
```

## Key

Kafka 发送消息时，key 可以不用指定。如果不指定消息的 key，则消息发送到的分区是随着时间不停变换的。如果指定了消息的 key，则会根据消息的 hash 值和 topic 的分区数取模来获取分区的。如果应用有消息顺序性的需要，则可以通过指定消息的 key 和自定义分区类来将符合某种规则的消息发送到同一个分区。同一个分区消息是有序的，同一个分区只有一个消费者就可以保证消息的顺序性消费。

```
1 log.segment.bytes=536870912
2 log.retention.check.interval.ms=60000
3 log.cleaner.enable=false
4 # 数据文件保留多长时间
5 # log.retention.bytes
6 # log.retention.minutes
7 # log.retention.hours 任意一个达到要求，都会执行删除
8 log.retention.minutes=300
9 log.retention.hours=24
```

#### 1.4.11 Swagger UI 使用总结

##### fetching resource list: null

Swagger UI 初始化时，可以显示首页，但是没有详细的注释，是由于相应的需要注解的 Controller 类没有添加 RestController 注解，没有扫描到目标 Controller。用户通过操作浏览器，向服务端发起 HTTP 请求，HTTP 请求通过 Web 容器提交到 Spring Web 的集中访问点 DispatcherServlet，由 DispatcherServlet 控制器查询一个或多个 HandlerMapping，找到处理请求的 Controller。

```
1 return new Docket(DocumentationType.SWAGGER_2)
2     .apiInfo(apiInfo())
3     .select()
4     .apis(RequestHandlerSelectors.withClassAnnotation(
5         RestController.class))
6     .paths(PathSelectors.any())
7     .build()
8     .globalOperationParameters(pars);
```

从配置可以看出，Swagger 是根据添加 RestController 注解 (Annotation) 来扫描相应的 Controller 并生成文档。客户端在发送 swagger-resources 请求时，获取到空的结果，原因是在 Swagger 配置中定义了重复的 ParameterBuilder 参数名称。

### 1.4.12 IntelliJ Idea 项目 \*.iml 文件

iml(Idea Module Level) 文件由 IntelliJ Idea 创建, 存储了模块 (Module) 的配置信息, 一般称之为模块文件 (Module File), 此文件默认存储在对应模块的根目录下。模块文件存储了模块配置, 模块内容和资源根目录, 依赖信息等等 (A module file (the .iml file) is used for keeping module configuration, including content or source roots, dependencies, framework-specific settings in facets, and so on.)<sup>19</sup>。当需要导入已经存在的模块时, 可以直接导入 .iml 文件。有时会出现 IntelliJ Idea 项目树上标识模块的蓝色的小图标变成灰色图标的情况 (蓝色部分消失)<sup>20</sup>, 那是由于 IntelliJ Idea 找不到模块的配置, 修复此问题, 可以在 .idea 目录下的文件 modules.xml 中调整项目的 Module 配置, 如果不存在 Module 配置, 添加配置, 如果名字不匹配, 调整具体的名字, 调整完毕后, 刷新项目蓝色图标即可自动恢复。

### 1.4.13 Gradle

#### Gradle 手动离线安装

在使用 gradle 命令下载 Gradle 文件时 (2019 年 08 月 08 日), 一直没有速度。可以先下载好 Gradle 安装包, 手动解压安装。使用 aria2c 工具下载好 Gradle 文件:

```
1 aria2c https://services.gradle.org/distributions/gradle-5.5.1-bin.  
    zip
```

如果在 Mac 下, 将下载好的文件拷贝到 Home 目录 .gradle/wrapper/dists 目录的对应 Gradle 版本的 bin 目录下, 最后一级目录类似 cfs0v38hb3r1zj4ic9bbjcc7n 随机字符串。拷贝 zip 包即可, 不用解压缩, 再次运行 Gradle 命令时, 会自动解压文件。

#### Gradle 代理

在 Home 目录下的 gradle 文件夹下新建 gradle.properties 文件:

```
1 allprojects{  
2     repositories {  
3         google()  
    }
```

<sup>19</sup>来源链接: <https://www.jetbrains.com/help/idea/creating-and-managing-modules.html>

<sup>20</sup>所有图标的含义见: <https://www.jetbrains.com/help/idea/symbols.html>。也可以在 IntelliJ Idea 的 Settings/Preferences dialog (.), go to | Editor | File Types 中查看, 不过仅仅是文件图标类型, 不包含文件夹图标类型。

```
4      def ALIYUN_REPOSITORY_URL = 'http://maven.aliyun.com/nexus
      /content/groups/public'
5      def ALIYUN_JCENTER_URL = 'http://maven.aliyun.com/nexus/
      content/repositories/jcenter'
6      all { ArtifactRepository repo ->
7          if(repo instanceof MavenArtifactRepository){
8              def url = repo.url.toString()
9              if (url.startsWith('https://repo1.maven.org/maven2
      ')) {
10                  project.logger.lifecycle "Repository ${repo.
      url} replaced by $ALIYUN_REPOSITORY_URL."
11                  remove repo
12              }
13              if (url.startsWith('https://jcenter.bintray.com/'))
      ) {
14                  project.logger.lifecycle "Repository ${repo.
      url} replaced by $ALIYUN_JCENTER_URL."
15                  remove repo
16              }
17          }
18      }
19      maven { url ALIYUN_REPOSITORY_URL }
20      maven { url ALIYUN_JCENTER_URL }
21
22  }
23 }
```

#### 1.4.14 Spring Bean 初始化顺序

在一次 Spring 的构造函数初始化 RabbitMQ 的消费者过程中，由 Apollo 属性注入的配置项获取的值为空，第一反应是与 Bean 的构造函数和属性的实例化顺序导致此问题，在研究了 Bean 的初始化过程后，清楚了是由于 Bean 构造函数先于 Bean 属性初始化，所以在构造函数中获取的属性配置为空。在真正执行创建 Bean 动作之

前, Spring 会尝试从当前容器或者父容器的缓存中获取 Bean。当在缓存中没有获取到 Bean 时, 就会执行创建 Bean 的流程。整个 IOC 容器的启动过程都包含在容器抽象类 `AbstractApplicationContext` 的模板方法 `refresh()` 中在这之前已经创建了核心容器 `BeanFactory`, 完成了 bean 定义信息的加载解析和注册, 对于用户定义的每一个 bean, 创建一个对应的 `BeanDefinition`, 以 `beanName` 为 key, `Definition` 为 value 保存在核心容器 `beanFactory` 的 map 中。这个时候还没有真正创建 Bean, 而是创建了一个 Bean 的设计图——`BeanDefinition`, 之后可以根据这个 `BeanDefinition` 创建真正的 Bean 实例。完成核心容器的创建后, 还会注册一些容器的基础组件, 之后才会来到启动容器最重要的阶段——初始化 bean 的阶段, 这部分的入口在 `finishBeanFactoryInitialization(beanFactory)` 方法中。创建 Bean 在类 `AbstractAutowireCapableBeanFactory` 中的 `createBean` 方法 (基于 5.1.7.RELEASE 版本), 而注入属性在类的第 218 行 `autowireBeanProperties` 方法。Spring Bean 初始化时, 首先初始化构造函数。初始化默认的无参构造函数在类 `AbstractAutowireCapableBeanFactory` 的 1193 行:

```
1 if (ctors != null) {  
2     return autowireConstructor(beanName, mbd, ctors, null);  
3 }
```

构造函数 (Constructor) 初始化完毕后, 初始化属性。初始化属性在方法 `populateBean` 中, 在类 `AbstractAutowireCapableBeanFactory` 的 592 行:

```
1 try {  
2     populateBean(beanName, mbd, instanceWrapper);  
3     exposedObject = initializeBean(beanName, exposedObject, mbd);  
4 }
```

如果实现了 `BeanFactoryAware` 接口执行 `setBeanFactory` 方法。如果实现了 `InitializingBean` 接口执行 `afterPropertiesSet` 方法。如果在配置文件中指定了 `init-method`, 那么执行该方法。如果实现了 `BeanFactoryPostProcessor` 接口在“new”其他类之前执行 `postProcessBeanFactory` 方法 (通过这个方法可以改变配置文件里面的属性值的配置)。如果实现了 `BeanPostProcessor` 接口, 那么会在其他 bean 初始化方法之前执行 `postProcessBeforeInitialization` 方法, 之后执行 `postProcessAfterInitialization` 方法

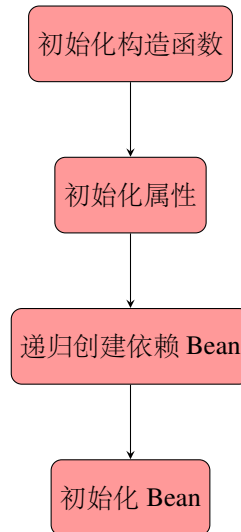


Figure 1.8: Spring Bean 初始化过程

#### 1.4.15 Spring 加载完成后执行

项目中需要初始化 MQ，由于配置是从 Apollo 中获取，Apollo 在初始化 Bean 时还未初始化，所以放在 Spring Boot 初始化完毕后运行：

```
1 @Slf4j
2 @Component
3 public class ApplicationStarted implements ApplicationListener<
4     ContextRefreshedEvent> {
5
6     @Autowired
7     private MessageQueueConsumer messageQueueConsumer;
8
9     @Override
10    public void onApplicationEvent(ContextRefreshedEvent event) {
11        try {
12            messageQueueConsumer.initMQ();
13            log.info("MQ初始化完成");
```

```
13         } catch (Exception e) {  
14             log.error("初始化MQ遇到错误", e);  
15         }  
16     }  
17 }
```

#### 1.4.16 Apollo 配置的加载时机

本地环境变量中的配置 > apollo 配置中心的配置 > 本地文件的配置，apollo 配置中心的配置在初始化 bean 之后，在 bean 的构造方法之后初始化。Spring Boot 启动时，经过前面的资源收集，然后加载 `ApplicationContextInitializer`，即执行 `ApolloApplicationContextInitializer.java` 中 `initialize()`。日志在 1.2.0 版本调整后放入 Apollo 中，但是需要在本地 `application.properties` 中添加如下配置：

```
1 apollo.bootstrap.enabled=true  
2 apollo.bootstrap.eagerLoad.enabled = true
```

#### 1.4.17 Jenkins 远程执行 Shell

在 Jenkins 中通过插件 `publish over ssh` 远程执行 Shell 时，Shell 脚本中的 `nohup` 指令始终不执行。通过反复尝试，发现在目标主机初始状态不存在需要启动的 Java 进程时，使用 Jenkins 远程执行 shell 的第一次无法执行 `nohup` 指令，再次远程通过插件 `publish over ssh` 执行 shell 脚本时能成功执行。目前还不知道为什么会出现这种情况。解决问题的方法是在 `nohup` 指令后添加 `sleep` 指令。如下代码片段所示：

```
1 count='ps -ef | grep ${APP_NAME} | grep -v "grep" | wc -l'  
2 if [[ ${count} -lt 1 ]]; then  
3     cd ${APP_HOME}  
4     nohup ${JAVA_HOME}/bin/java -Xmx256M -Xms128M -jar -Xdebug -  
        Xrunjdwp:transport=dt_socket,suspend=n,server=y,address  
        =5016 ${APP_HOME}/${APP_NAME} >> /dev/null &  
5     sleep 5  
6 else  
7     echo "process already exists!"
```

```
8   exit 1
9 fi
```

最终的原因发现是因为 `nohup` 命令需要片刻时间来启动 `Command` 参数指定的命令，在注销前需要等待一定时间。如果太快注销，`Command` 参数指定的命令可能根本没运行。一旦 `Command` 参数指定的命令启动，注销就不会对其产生影响<sup>21</sup>。而在远程通过 `Jenkins` 插件 `publish over ssh` 执行脚本时，就没有等待的时间，执行完毕立即退出，所以造成最后一条 `nohup` 指令没有执行到。

#### 1.4.18 消息队列通用的消息持久化

目标是在消息队列的消费端持久化不同的实体只需要添加配置即可，不需要改动代码。定义消息发送结构：

```
1 {
2     "message_type": "user_statistic", //持久化时消息类型与数据表关联
        存储
3     "message_id": "uniq_id", //去重
4     "client_info": //生产上报数据微服务信息
5     {
6         //client info
7     },
8     "device_info": //产生日志设备信息
9     {
10    },
11    "extra_info": //扩展信息
12    {
13    },
14    "data": //主要信息
15    {
16        //content
17    }
```

<sup>21</sup>来源链接: [https://www.ibm.com/support/knowledgecenter/zh/ssw\\_aix\\_72/com.ibm.aix.cmds4/nohup.htm%0A](https://www.ibm.com/support/knowledgecenter/zh/ssw_aix_72/com.ibm.aix.cmds4/nohup.htm%0A)



```
18 }
```

消息唯一 ID 用于消息去重判断，保持消息的幂等。数据库根据消息类型配置对应的数据表，接收到消息时，根据消息类型找到对应的数据表名称，并根据名称和列生成动态 SQL，生成动态 SQL 时数据表的列可以从数据库中查询，最后利用拼接的动态 SQL 将消息持久化到数据库。这样在增删改消息时，只需要变动配置信息，不再修改任何一行代码，理论上在不该动任何一行代码的情况下可以适配所有的消息类型。由于埋点日志消息在每个微服务中都需要使用，日志消息设计一个统一的日志微服务，日志微服务开启熔断机制，在日志阻塞或者服务中断时，不影响主要业务流程的继续进行。所有服务的日志格式为标准的 JSONObject，如：

```
1 JSONObject obj = new JSONObject();
2 obj.put("action","user");
3 obj.put("user_id","i1");
4 logs.add(obj);
```

#### 1.4.19 Kibana 使用心得

过滤 Websocket 推送消息：

```
1 log.file.path : /data/logs/ws-red-envelope/spring.log and message
   : session
```

精确匹配内容，需要在内容上加上双引号：

```
1 message : "缓存与数据库不一致"
```

Kibana 7.2.0 默认的时间过滤是最新的 15 分钟，但是日志分析的习惯是结合最近几天来分析，调整默认的时间过滤器范围，在 Kibana->Advanced Settings>Time picker defaults，设置为：

```
1 {
2   "from": "now-10d",
3   "to": "now"
```

```
4 }
```

### 1.4.20 函数式接口之一——Predicate

`java.util.function.Predicate` 是在 Java 8 中引入的函数式接口 (Functional Interface), 它接受一个输入参数 `T`, 返回一个布尔值结果。该接口包含多种默认方法来将 `Predicate` 组合成其他复杂的逻辑 (比如: 与, 或, 非)。该接口用于测试对象是 `true` 或 `false`。例如, 可以将不同 Lambda 表达式的条件通过 `Predicate` 定义后通过参数传递:

```
1 Predicate<RoomSeatResponse> robotCondition = item -> {  
2     return item.getCount() >= 10 && item.getCount() / 20.0 < 0.8;  
3 };  
4 roomSeat = matchImpl(userId, robotCondition);
```

### 1.4.21 Java 8 Stream

手动抛出自定义异常:

```
1 App app = apps.stream()  
2     .filter(filter)  
3     .findAny()  
4     .orElseThrow(() -> {  
5         throw AnalysisException.DATA_EXCEPTION;  
6     });
```

此种写法在用 Maven 编译插件 (插件版本 3.8.0) 编译时, 提示错误: `unreported exception java.lang.Throwable; must be caught or declared to be thrown`。这个是由于 `javac` 编译器的类型隐式推断的 Bug<sup>22</sup>, Bug 的编号是 `JDK-8054569`。但是当前项目的 Java 版本是 1.8.0 修改版本 211, 不会出现此问题才是, 不过也有升级版本后依然存在此问题。目前解决此问题的方法之一是显示指定类型<sup>23</sup>, 写法如下:

```
1 App app = apps.stream()
```

<sup>22</sup>信息来源: <https://gist.github.com/rompetroll/667bf46ac0168a92497a>

<sup>23</sup>参考来源: <https://stackoverflow.com/questions/25523375/java8-lambdas-and-exceptions>

```
2         .filter(filter)
3         .findAny()
4         .<AnalysisException>orElseThrow(() -> {
5             throw AnalysisException.DATA_EXCEPTION;
6         });
```

### 1.4.22 MySQL 常见操作

从 information\_schema.innodb\_trx 表中查看当前未提交的事务:

```
1 select trx_id, trx_state, trx_started, trx_mysql_thread_id,
   trx_query
2 from information_schema.innodb_trx;
```

trx\_mysql\_thread\_id: MySQL 的线程 ID, 用于 kill。登陆:

```
1 mysql -u root -p
```

输入密码即可。登陆提示 MySQL Access denied for user 'root'@'IP 地址', 执行如下语句 (不安全, 正式环境需按照安全要求设置):

```
1 GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY '123456'
   WITH GRANT OPTION;
2 FLUSH PRIVILEGES;
```

根据 A 表更新 B 表:

```
1 update robot r, u_user u
2 set r.'user_mark' = u.'user_mark'
3 where r.'user_id' = u.'id'
```

### 1.4.23 top 命令

显示 command 详细的输出:

```
1 top -c
```

按小写的 **e**，可以将列表输出的单位在 **KB/MB/GB/TB/PB** 之间切换。**ctrl + A** 可以使用键盘的左右方向键查看没有显示完全的详细命令内容。**shift + m** 可以按内存进行排序。有时排查问题时，需要将 **top** 输出冻结较长时间，使用 **d** 参数：

```
1 # 每 10s 刷新一次
2 top -d 10
```

#### 1.4.24 Grep 使用小技巧

看看日志关键字上下文：

```
1 # 打印匹配关键字的前后 3 行记录
2 tail -f soa-robot.log | grep -C 3 "提交成绩"
```

查看日志关联前后文：

```
1 # 查看用户报名和提交成绩
2 # 匹配成绩或报名关键字
3 tail -n 2000 -f soa-robot.log | grep -E "提交成绩|报名"
```

#### 1.4.25 空指针异常

在使用 **HashMap** 里面的对象方法时，要做非常多的非空判断：

```
1 for (Map.Entry<Long, ClientSession> entry : SessionUtil.
    websocketSessions.entrySet()) {
2     if (entry.getValue() != null && entry.getValue().getSession()
        != null && entry.getValue().getSession().getId().equals(
        session.getId())) {
3         currentTargetScore = entry.getValue().getTickTime();
4         break;
```

```
5     }  
6 }
```

使用 Optional，代码简洁许多：

```
1 for (Map.Entry<Long, ClientSession> entry : SessionUtil.  
    websocketSessions.entrySet()) {  
2     String sessionId = Optional.of(entry.getValue())  
3         .map(value -> value.getSession())  
4         .map(value -> value.getId())  
5         .orElse("");  
6     if (sessionId.equals(session.getId())) {  
7         currentTargetScore = entry.getValue().getRoundId();  
8         break;  
9     }  
10 }
```

### 1.4.26 Kubernetes 部署

部署 Kubernetes 最难的点是获取部署文件，这个问题解决了，其他问题就都顺利了。Kubernetes 是 Google 基于 Borg 开源的容器编排调度引擎，作为 CNCF（Cloud Native Computing Foundation）最重要的组件之一，它的目标不仅仅是一个编排系统，而是提供一个规范，可以让你来描述集群的架构，定义服务的最终状态，Kubernetes 可以帮你将系统自动地达到和维持在这个状态。安装步骤来源于 handbook<sup>24</sup>。后来发现 Kubernetes 官方文档<sup>25</sup>描述更详细、更清晰。还可以参考 Kubernetes 交互式部署手册<sup>26</sup>。

定义整个 kubernetes 集群都需要使用到的环境变量 (Environment Variable)<sup>27</sup>：

<sup>24</sup>参考地址：<https://jimmysong.io/kubernetes-handbook/practice/create-tls-and-secret-key.html>

<sup>25</sup><https://kubernetes.io/zh/docs/concepts/cluster-administration/certificates/>

<sup>26</sup><https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-interactive/>

<sup>27</sup>来源地址：<https://github.com/opsnull/follow-me-install-kubernetes-cluster>

Table 1.4: 组件分布

主机	角色	组件
172.19.104.231	k8s master	etcd, kube-apiserver, kube-controller-manager, kube-scheduler
172.19.104.230	k8s node01	docker, kubelet, kube-proxy
172.19.150.82	k8s node02	docker, kubelet, kube-proxy

```

1  #!/usr/bin/bash
2
3  # 生成 EncryptionConfig 所需的加密 key
4  export ENCRYPTION_KEY=$(head -c 32 /dev/urandom | base64)
5
6  # 集群各机器 IP 数组
7  export NODE_IPS=(172.19.104.231 172.19.104.230 172.19.150.82)
8
9  # 集群各 IP 对应的主机名数组
10 # Azshara 是游戏《魔兽世界》中的角色
11 export NODE_NAMES=(azshara-k8s01 azshara-k8s02 azshara-k8s03)
12
13 # etcd 集群服务地址列表
14 export ETCD_ENDPOINTS="https://172.19.104.231:2379,https
    ://172.19.104.230:2379,https://172.19.150.82:2379"
15
16 # etcd 集群间通信的 IP 和端口
17 export ETCD_NODES="azshara-k8s01=https://172.19.104.231:2380,
    azshara-k8s02=https://172.19.104.230:2380,azshara-k8s03=
    https://172.19.150.82:2380"
18
19 # kube-apiserver 的反向代理 (kube-nginx) 地址端口
20 export KUBE_APISERVER="https://127.0.0.1:8443"
21

```

```
22 # 节点间互联网络接口名称
23 export IFACE="eth0"
24
25 # etcd 数据目录
26 export ETCD_DATA_DIR="/opt/k8s/etcd/data"
27
28 # etcd WAL 目录, 建议是 SSD 磁盘分区
29 # 或者和 ETCD_DATA_DIR 不同的磁盘分区
30 export ETCD_WAL_DIR="/opt/k8s/etcd/wal"
31
32 # k8s 各组件数据目录
33 export K8S_DIR="/opt/k8s/k8s"
34
35 # docker 数据目录
36 export DOCKER_DIR="/opt/k8s/docker"
37
38 # 以下参数一般不需要修改
39
40 # TLS Bootstrapping 使用的 Token
41 # 可以使用命令 head -c 16 /dev/urandom|od -An -t x|tr -d ' ' 生成
42 BOOTSTRAP_TOKEN="41f7e4ba8b7be874fcff18bf5cf41a7c"
43
44 # 最好使用当前未用的网段来定义服务网段和 Pod 网段
45
46 # 服务网段, 部署前路由不可达, 部署后集群内路由可达 (kube-proxy 保证)
47 SERVICE_CIDR="10.254.0.0/16"
48
49 # Pod 网段, 建议/16 段地址
50 # 部署前路由不可达, 部署后集群内路由可达 (flanneld 保证)
51 CLUSTER_CIDR="172.30.0.0/16"
52
53 # 服务端口范围 (NodePort Range)
54 export NODE_PORT_RANGE="30000-32767"
```

```
55
56 # flanneld 网络配置前缀
57 export FLANNEL_ETCD_PREFIX="/kubernetes/network"
58
59 # kubernetes 服务 IP(一般是 SERVICE_CIDR 中第一个 IP)
60 export CLUSTER_KUBERNETES_SVC_IP="10.254.0.1"
61
62 # 集群 DNS 服务 IP(从 SERVICE_CIDR 中预分配)
63 export CLUSTER_DNS_SVC_IP="10.254.0.2"
64
65 # 集群 DNS 域名 (末尾不带点号)
66 export CLUSTER_DNS_DOMAIN="cluster.local"
67
68 # 将二进制目录/opt/k8s/bin 加到 PATH 中
69 export PATH=/opt/k8s/bin:$PATH
```

这里设置数据目录时,如果是多主机挂载的同一个磁盘,那么数据目录就不能够放在这个挂载的磁盘下,原因很简单,后面等于多个主机的数据放在一个目录下,会冲突,除非确定这个数据只有一个主机会用到。因为用阿里云挂载的数据盘很大,开始时,将 etcd 的数据文件放在 data 目录(data 目录挂载的一个 500GB 的数据盘)下,深深的为自己的高瞻远瞩和机制折服,后面 etcd 启动的时候总是提示文件锁定,更奇怪的是启动了 a 节点, b 节点宕掉了,百撕不得其解。还好反应过来了,否则不知道要掉坑里多久。调整环境变量中的路径,重新运行脚本生成服务即可。

### 创建 TLS 证书和秘钥

#### 安装证书创建工具 CFSSL

生成证书可以通过 easyrsa、openssl 或 cfssl 生成证书。创建传输层安全 TLS(Transport Layer Security) 证书和密钥在服务器 172.19.104.231 上进行。直接使用二进制源码包安装 CFSSL(CloudFlare Secure Sockets Layer):

```
1 wget https://pkg.cfssl.org/R1.2/cfssl_linux-amd64
2 chmod +x cfssl_linux-amd64
3 mv cfssl_linux-amd64 /usr/local/bin/cfssl
4
```



```
5 wget https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64
6 chmod +x cfssljson_linux-amd64
7 mv cfssljson_linux-amd64 /usr/local/bin/cfssljson
8
9 wget https://pkg.cfssl.org/R1.2/cfssl-certinfo_linux-amd64
10 chmod +x cfssl-certinfo_linux-amd64
11 mv cfssl-certinfo_linux-amd64 /usr/local/bin/cfssl-certinfo
12
13 git clone https://github.com/cloudflare/cfssl.git $GOPATH/src/
    github.com/cloudflare/cfssl
14
15 export PATH=/usr/local/bin:$PATH
```

遗憾的是，1.2 版本貌似有 bug<sup>28</sup>。在后续使用 1.2 版本 cfssl 工具生成 Kubernetes 证书时提示错误：This certificate lacks a "hosts" field. This makes it unsuitable for websites. For more information see the Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates, v.1.1.6, from the CA/Browser Forum (<https://cabforum.org>); specifically, section 10.2.3 ("Information Requirements"). 避免此问题，可升级 cfssl 工具的版本，使用如下命令升级：

```
1 # 下载 binary 包
2 wget -c https://dl.google.com/go/go1.12.9.linux-amd64.tar.gz
3 # 安装 go
4 tar -C /usr/local -xzf go1.12.9.linux-amd64.tar.gz
5 # 升级 cfssl 到最新版本
6 # 此命令下载、编译、安装 cfssl 工具
7 # 安装路径在 GOPATH/bin/cfssl
8 /usr/local/go/bin/go get -u github.com/cloudflare/cfssl/cmd/cfssl
9 # 查看 cfssl 版本
10 /root/go/bin/cfssl version
```

---

<sup>28</sup>参考 Issue: <https://github.com/cloudflare/cfssl/issues/717>

需要注意到细节是，截至到 2019 年 8 月 25 日，采用 go 命令安装 cfssl 需要 go 到版本是 1.12+，使用默认的 yum 仓库的 go 之前，预先看一看版本是否符合要求。

### 生成 CA 证书和私钥

CA 证书是根证书 (Root Certificate)，一般网站都希望用户知道他们建立的网络通道是安全的，所以会向 CA 机构购买证书来验证 domain，所以我們也可以在很多 HTTPS 的网页地址栏看到一把小绿锁。然而在一些情况下，我们没必要去 CA 机构购买证书，比如在内网的测试环境中，在此次内网部署 Kubernetes 集群中，为了验证 HTTPS 下的一些问题，不需要部署昂贵的证书，这个时候自建 Root CA，给自己颁发证书就可以搞事情<sup>29</sup>。

### 创建 CA 根证书配置文件

输入如下命令创建 CA 根证书 (Root Certificate) 配置文件：

```
1 mkdir /root/ssl
2 cd /root/ssl
3 cfssl print-defaults config > config.json
4 cfssl print-defaults csr > csr.json
5 # 根据config.json文件的格式创建如下的ca-config.json文件
6 # 过期时间设置成了 87600h
7 cat > ca-config.json <<EOF
8 {
9     "signing": {
10         "default": {
11             "expiry": "87600h"
12         },
13         "profiles": {
14             "kubernetes": {
15                 "usages": [
16                     "signing",
17                     "key encipherment",
18                     "server auth",
```

<sup>29</sup>文字来源小胡子哥的博客：<https://www.barretlee.com/blog/2016/04/24/detail-about-ca-and-certs/>。就我所知，目前除了阮一峰老师、陈浩老师的博客，小胡子的博客文章质量上乘。

```
19         "client auth"
20     ],
21     "expiry": "87600h"
22 }
23 }
24 }
25 }
26 EOF
```

- signing: 表示该证书可用于签名其它证书；生成的 ca.pem 证书中 CA=TRUE；
- server auth: 表示 client 可以用该 CA 对 server 提供的证书进行验证；
- client auth: 表示 server 可以用该 CA 对 client 提供的证书进行验证；

### 创建 CA 证书签名请求

创建 ca-csr.json 文件，内容如下：

```
1 {
2     "CN": "kubernetes",
3     "hosts": [],
4     "key": {
5         "algo": "rsa",
6         "size": 2048
7     },
8     "names": [
9         {
10             "C": "CN",
11             "ST": "BeiJing",
12             "L": "BeiJing",
13             "O": "k8s",
14             "OU": "System"
15         }
16     ],
17     "ca": {
```

```
18     "expiry": "87600h"
19   }
20 }
```

algo 是算法 (algorithm) 的缩写, 表示 CA 证书使用的什么加密算法, 这里指定的是公钥加密算法, 也叫做非对称加密算法, RSA 是设计算法的三位数学家 Rivest、Shamir 和 Adleman 名字英文首字母缩写。"CN": Common Name, kube-apiserver 从证书中提取该字段作为请求的用户名 (User Name); 浏览器使用该字段验证网站是否合法; "O": Organization, kube-apiserver 从证书中提取该字段作为请求用户所属的组 (Group); 输入如下命令生成证书和私钥:

```
1 cfssl gencert -initca ca-csr.json | cfssljson -bare ca
```

生成的文件如下表所示:

Table 1.5: CA 证书文件列表

文件名称	作用	备注
ca-config.json	CA 证书配置文件	
ca.csr	CA 证书	
ca-csr.json	CA 证书签名请求	
ca-key.pem	CA 证书私钥	
ca.pem	CA 证书公钥	

创建 Kubernetes 证书

创建证书的操作在 Master 主机上执行, 这里使用的 Master 主机是 172.19.104.231。创建 Kubernetes 证书签名请求 (Certificate Sign Request) 文件 kubernetes-csr.json:

```
1 {
2   "CN": "kubernetes",
3   "hosts": [
4     "127.0.0.1",
5     "172.19.104.230",
```

```
6     "172.19.150.82",
7     "172.19.104.231"
8 ],
9 "key": {
10     "algo": "rsa",
11     "size": 2048
12 },
13 "names": [
14     {
15         "C": "CN",
16         "ST": "BeiJing",
17         "L": "BeiJing",
18         "O": "k8s",
19         "OU": "System"
20     }
21 ]
22 }
```

172.19.104.230 为 Master IP, 172.19.150.82 为 Cluster Master IP, 是 API 服务器的服务集群 IP。运行如下命令生成 Kubernetes 证书和私钥, 签发证书使用的是上一步生成的根证书 (Root Certificate):

```
1 /root/go/bin/cfssl gencert -ca=ca.pem -ca-key=ca-key.pem \
2 -config=ca-config.json \
3 -profile=kubernetes \
4 kubernetes-csr.json | cfssljson -bare kubernetes
```

### 创建 admin 证书

创建 admin 证书签名请求文件 admin-csr.json:

```
1 {
2     "CN": "admin",
3     "hosts": [],
```

```
4  "key": {  
5      "algo": "rsa",  
6      "size": 2048  
7  },  
8  "names": [  
9      {  
10         "C": "CN",  
11         "ST": "BeiJing",  
12         "L": "BeiJing",  
13         "O": "system:masters",  
14         "OU": "System"  
15     }  
16 ]  
17 }
```

生成 admin 证书和私钥:

```
1 cfssl gencert -ca=ca.pem -ca-key=ca-key.pem \  
2 -config=ca-config.json \  
3 -profile=kubernetes admin-csr.json | cfssljson -bare admin
```

### 创建 kube-proxy 证书

创建 kube-proxy 证书签名请求文件 kube-proxy-csr.json:

```
1 {  
2     "CN": "system:kube-proxy",  
3     "hosts": [],  
4     "key": {  
5         "algo": "rsa",  
6         "size": 2048  
7     },  
8     "names": [  
9         {
```

```
10     "C": "CN",
11     "ST": "BeiJing",
12     "L": "BeiJing",
13     "O": "k8s",
14     "OU": "System"
15   }
16 ]
17 }
```

生成 kube-proxy 客户端证书和私钥:

```
1 cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.json
  \
2 -profile=kubernetes kube-proxy-csr.json | cfssljson -bare kube-
  proxy
```

将生成的证书和秘钥文件（后缀名为.pem）拷贝到所有机器的 /etc/kubernetes/ssl 目录下备用，PEM(Privacy Enhanced Mail) 是 X.509 其中的一种编码格式，是文本格式，以"—BEGIN..." 开头，"—END..." 结尾，内容是 BASE64 编码。DER - Distinguished Encoding Rules，是二进制格式，不可读。虽然有 PEM 和 DER 这两种编码格式，但文件扩展名并不一定就叫"PEM" 或者"DER"，常见的扩展名除了 PEM 和 DER 还有 CRT、CER、KEY、CSR、PFX/P12、JKS，它们除了编码格式可能不同之外，内容也有差别，但大多数都能相互转换编码格式。

客户端节点可能拒绝承认自签名 CA 证书有效。对于非生产环境的部署，或运行在企业防火墙后的部署，用户可以向所有客户端分发自签名 CA 证书，并刷新本地的有效证书列表。在每个客户端上执行以下操作（注意不同的操作系统类型或者版本执行的脚本不同，此处使用的操作系统是 CentOS 7.6）:

```
1 openssl x509 -outform der -in ca.pem -out ca.crt
2 # 拷贝 CA 根证书
3 cp /data/k8s/ssl/ca.crt /etc/pki/ca-trust/source/anchors/
4 update-ca-trust
```

### 安装 kubectl 命令行工具

kubectl 可以操控 Kubernetes 集群，安装 kubectl(Kubernetes Control) 命令行工具在 master 节点上操作。安装 kubectl 需要对应版本或者是高版本，尽量避免使用过低版本的 client，避免遇到兼容性等不可预知的问题，这里部署的 kubernetes 是 v1.15 版本<sup>30</sup>。查看当前最新的稳定版本的 kubectl 版本号：

```
1 curl -s https://storage.googleapis.com/kubernetes-release/release/  
    stable.txt
```

下载 kubectl，注意这里的下载需要通过代理方式，直接访问是无法下载的：

```
1 curl -LO https://storage.googleapis.com/kubernetes-release/  
    /v1.15.2/bin/linux/amd64/kubectl
```

由于在终端中无法下载 kubectl，浪费了不少时间，直接拷贝到浏览器下载即可，下载下来就一个 kubectl 文件。

### 创建 kubeconfig 文件

kubernetes 1.4 开始支持由 kube-apiserver 为客户端生成 TLS 证书的 TLS Bootstrapping 功能，这样就不需要为每个客户端生成证书了；该功能当前仅支持为 kubelet 生成证书；以下操作只需要在 master 节点上执行，生成的 \*.kubeconfig 文件可以直接拷贝到 node 节点的/etc/kubernetes 目录下。

### 创建 TLS Bootstrapping Token

Token 可以是任意的包含 128 bit 的字符串，可以使用安全的随机数发生器生成。使用如下命令来完成：

```
1 export BOOTSTRAP_TOKEN=$(head -c 16 /dev/urandom | od -An -t x |  
    tr -d ' ' )  
2 cat > token.csv <<EOF  
3 ${BOOTSTRAP_TOKEN},kubelet-bootstrap,10001,"system:kubelet-  
    bootstrap"  
4 EOF
```

<sup>30</sup>kubectl 官方安装文档：<https://kubernetes.io/docs/tasks/tools/install-kubectl/>



### 创建 kubelet bootstrapping kubeconfig 文件

安装 kubectl 后，执行如下命令：

```
1 cd /etc/kubernetes
2 # APISERVER 就是 master 节点
3 # 此处的 master 节点 IP 是 172.19.104.231
4 export KUBE_APISERVER="https://172.19.104.231:6443"
5
6 # 设置集群参数
7 kubectl config set-cluster kubernetes \
8     --certificate-authority=/etc/kubernetes/ssl/ca.pem \
9     --embed-certs=true \
10    --server=${KUBE_APISERVER} \
11    --kubeconfig=bootstrap.kubeconfig
12
13 # 设置客户端认证参数
14 kubectl config set-credentials kubelet-bootstrap \
15     --token=${BOOTSTRAP_TOKEN} \
16     --kubeconfig=bootstrap.kubeconfig
17
18 # 设置上下文参数
19 kubectl config set-context default \
20     --cluster=kubernetes \
21     --user=kubelet-bootstrap \
22     --kubeconfig=bootstrap.kubeconfig
23
24 # 设置默认上下文
25 kubectl config use-context default --kubeconfig=bootstrap.
    kubeconfig
```

### 创建 kube-proxy kubeconfig 文件

```
1 export KUBE_APISERVER="https://172.19.104.231:6443"
```

```
2 # 设置集群参数
3 kubectl config set-cluster kubernetes \
4   --certificate-authority=/etc/kubernetes/ssl/ca.pem \
5   --embed-certs=true \
6   --server=${KUBE_APISERVER} \
7   --kubeconfig=kube-proxy.kubeconfig
8 # 设置客户端认证参数
9 kubectl config set-credentials kube-proxy \
10  --client-certificate=/etc/kubernetes/ssl/kube-proxy.pem \
11  --client-key=/etc/kubernetes/ssl/kube-proxy-key.pem \
12  --embed-certs=true \
13  --kubeconfig=kube-proxy.kubeconfig
14 # 设置上下文参数
15 kubectl config set-context default \
16  --cluster=kubernetes \
17  --user=kube-proxy \
18  --kubeconfig=kube-proxy.kubeconfig
19 # 设置默认上下文
20 kubectl config use-context default --kubeconfig=kube-proxy.
    kubeconfig
```

生成的 kubeconfig 被保存到 `/.kube/config` 文件，`/.kube/config` 文件拥有对该集群的最高权限，需要妥善保管。

### 创建高可用 etcd 集群

Kubernetes 系统使用 etcd 存储所有数据，etcd 是 CoreOS 团队于 2013 年 6 月发起的开源项目，它的目标是构建一个高可用的分布式键值 (key-value) 数据库。这个词的发音是“et-see-dee”，表示在多台机器上分发 Unix 系统的“/etc”目录，其中包含了大量的全局配置文件。它是许多分布式系统的主干，为跨服务器集群存储数据提供可靠的方式。它适用于各种操作系统，包括 Linux、BSD 和 OS X。etcd 内部采用 raft 协议作为一致性算法，etcd 基于 Go 语言实现。

Table 1.6: etcd 组件分布

主机	角色	组件
172.19.104.231	Master	etcd 集群 Leader 节点
172.19.104.230	Worker	etcd 集群 Follower 节点
172.19.150.82	Worker	etcd 集群 Follower 节点

### 移除旧文件

后续有一个节点无法加入集群，为了熟悉 etcd，再重新安装一遍，重新安装之前删除旧的数据：

```
1 cd /var/lib/etcd
```

### 下载二进制文件

部署 etcd 3.3.13：

```
1 # 下载 etcd 二进制包
2 wget -c https://github.com/coreos/etcd/releases/download/v3.3.13/
   etcd-v3.3.13-linux-amd64.tar.gz
3 tar -zxvf etcd-v3.3.13-linux-amd64.tar.gz
```

将二进制文件分发到集群的各个节点，注意如果目标集群没有对应文件夹，新建文件夹即可，当前操作的节点与其他节点要做免密登录：

```
1 cd /opt/k8s/work
2 source /opt/k8s/bin/environment.sh
3 for node_ip in ${NODE_IPS[@]}
4 do
5     echo ">>> ${node_ip}"
6     scp etcd-v3.3.13-linux-amd64/etcd* root@${node_ip}:/opt/k8s/
       bin
7     ssh root@${node_ip} "chmod +x /opt/k8s/bin/*"
```

8 done

### 创建 etcd 证书和私钥

创建证书签名请求 (Certificate Sign Request):

```
1 cd /opt/k8s/work
2 cat > etcd-csr.json <<EOF
3 {
4     "CN": "etcd",
5     "hosts": [
6         "127.0.0.1",
7         "172.19.104.231",
8         "172.19.104.230",
9         "172.19.150.82"
10    ],
11    "key": {
12        "algo": "rsa",
13        "size": 2048
14    },
15    "names": [
16        {
17            "C": "CN",
18            "ST": "BeiJing",
19            "L": "BeiJing",
20            "O": "k8s",
21            "OU": "4Paradigm"
22        }
23    ]
24 }
25 EOF
```

生成证书和私钥:

```
1 cd /opt/k8s/work
2 /root/go/bin/cfssl gencert -ca=/opt/k8s/work/ca.pem \
3   -ca-key=/opt/k8s/work/ca-key.pem \
4   -config=/opt/k8s/work/ca-config.json \
5   -profile=kubernetes etcd-csr.json | cfssljson -bare etcd
6 ls etcd*pem
```

分发生成的证书和私钥到各 etcd 节点:

```
1 cd /opt/k8s/work
2 source /opt/k8s/bin/environment.sh
3 for node_ip in ${NODE_IPS[@]}
4 do
5   echo ">>> ${node_ip}"
6   ssh root@${node_ip} "mkdir -p /etc/etcd/cert"
7   scp etcd*.pem root@${node_ip}:/etc/etcd/cert/
8 done
```

### 创建和分发 etcd 的 systemd unit 文件

利用创建的 system Unit 文件模版, 执行如下脚本生成各个节点的 system unit 文件, 在/usr/lib/systemd/system/目录下创建文件 etcd.service, 内容如下。注意替换 IP 地址为你自己的 etcd 集群的主机 IP。

```
1 cd /opt/k8s/work
2 source /opt/k8s/bin/environment.sh
3 for (( i=0; i < 3; i++ ))
4 do
5   sed -e "s/##NODE_NAME##/${NODE_NAMES[i]}/" -e "s/##NODE_IP##/${
6     NODE_IPS[i]}/" etcd.service.template > etcd-${NODE_IPS[i]
7     }.service
8 done
```

环境变量配置文件/etc/etcd/etcd.conf:

```
1 # [member]
2 ETCD_NAME=infra1
3 ETCD_DATA_DIR="/var/lib/etcd"
4 ETCD_LISTEN_PEER_URLS="https://172.19.150.82:2380"
5 ETCD_LISTEN_CLIENT_URLS="https://172.19.150.82:2379"
6
7 # [cluster]
8 ETCD_INITIAL_ADVERTISE_PEER_URLS="https://172.19.150.82:2380"
9 ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
10 ETCD_ADVERTISE_CLIENT_URLS="https://172.19.150.82:2379"
```

### 启动 etcd 服务

如果有用其他方式安装过 etcd，启动时可能提示文件存在，删除之前创建的链接即可：

```
1 cd /opt/k8s/work
2 source /opt/k8s/bin/environment.sh
3 for node_ip in ${NODE_IPS[@]}
4 do
5     echo ">>> ${node_ip}"
6     ssh root@${node_ip} "mkdir -p ${ETCD_DATA_DIR} ${ETCD_WAL_DIR}
7     "
8     ssh root@${node_ip} "systemctl daemon-reload && systemctl
9     enable etcd && systemctl restart etcd " &
10 done
```

启动之后，检查当前节点的状态：

```
1 # 查看状态
2 /opt/k8s/bin/etcdctl endpoint health
3 # 启动状态日志
```

```
4 systemctl status etcd.service
```

在所有的 Kubernetes Master 节点重复上面的步骤 (节点 1:172.19.104.231, 节点 2:172.19.104.230, 节点 3:172.19.150.82), 直到所有机器的 etcd 服务都已启动。Kubernetes 的 master 服务主要包括 etcd(数据存储)、control-manager(控制器)、scheduler(调度器)、apiserver(服务接口), 我们将其部署到多节点实现容错。etcd 服务启动失败时, 使用如下命令查看日志:

```
1 journalctl -xe
```

journal 是 systemd 的日志系统, 是 systemd 的一部分, 从而使得 systemd 不依赖于其他组件, 例如不用等待 syslog 守护进程。启动时提示 Failed at step CHDIR spawning /usr/local/bin/etcd: No such file or directory 错误, 原因是/var/lib/etcd 这个目录不存在, 创建该目录即可。如果无法以服务的方式启动, 在机器 172.19.104.231 机器上, 尝试通过命令行方式启动 etcd 的 infra1 节点:

```
1 /usr/local/bin/etcd \  
2   --name infra1 \  
3   --cert-file=/etc/kubernetes/ssl/kubernetes.pem \  
4   --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \  
5   --peer-cert-file=/etc/kubernetes/ssl/kubernetes.pem \  
6   --peer-key-file=/etc/kubernetes/ssl/kubernetes-key.pem \  
7   --trusted-ca-file=/etc/kubernetes/ssl/ca.pem \  
8   --peer-trusted-ca-file=/etc/kubernetes/ssl/ca.pem \  
9   --initial-advertise-peer-urls https://172.19.104.231:2380 \  
10  --listen-peer-urls https://172.19.104.231:2380 \  
11  --listen-client-urls https://127.0.0.1:2379 \  
12  --advertise-client-urls https://172.19.104.231:2379 \  
13  --initial-cluster-token etcd-cluster \  
14  --initial-cluster infra1=https://172.19.104.231:2380,infra2=  
    https://172.19.104.230:2380,infra3=https  
    ://172.19.150.82:2380 \  
15  --initial-cluster-state new \  

```

```
16 --data-dir=/var/lib/etcd
```

- `listen-peer-urls` 是表示 `etcd` 和同伴 (`peer`) 通信的地址, 多个使用逗号分隔, 注意要使得所有节点都能访问, 不要使用 `localhost` 作为配置的 URL 的主机标记。
- `listen-client-urls` 对外提供服务的地址: 比如 `http://ip:2379,http://127.0.0.1:2379`, 客户端会连接到这里和 `etcd` 交互
- `advertise-client-urls` 是 `etcd` 客户端使用, 客户端通过该地址与本 `member` 交互信息。一定要保证从客户侧能可访问该地址。
- `initial-advertise-peer-urls`: 其他 `member` 使用, 其他 `member` 通过该地址与本 `member` 交互信息。一定要保证从其他 `member` 能可访问该地址。静态配置方式下, 该参数的 `value` 一定要同时在 `-initial-cluster` 参数中存在。`memberID` 的生成受 `-initial-cluster-token` 和 `-initial-advertise-peer-urls` 影响。

`etcd` 目前默认使用 2379 端口提供 HTTP API 服务, 2380 端口和 `peer` 通信 (这两个端口已经被 IANA 官方预留给 `etcd`)。启动后使用命令 `etcdctl member list` 查看 `etcd` 集群状态时提示: `x509: certificate signed by unknown authority`。解决此问题, 指定 `etcdctl` 版本:

```
1 export ETCDCTL_API=3
```

启动节点 1:

```
1 /usr/local/bin/etcd \  
2 --name infra3 \  
3 --cert-file=/etc/kubernetes/ssl/kubernetes.pem \  
4 --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \  
5 --peer-cert-file=/etc/kubernetes/ssl/kubernetes.pem \  
6 --peer-key-file=/etc/kubernetes/ssl/kubernetes-key.pem \  
7 --trusted-ca-file=/etc/kubernetes/ssl/ca.pem \  
8 --peer-trusted-ca-file=/etc/kubernetes/ssl/ca.pem \  
9 --initial-advertise-peer-urls https://172.19.150.82:2380 \  
10 --listen-peer-urls https://172.19.150.82:2380 \  
11 --listen-client-urls http://127.0.0.1:2379 \  
12 --advertise-client-urls https://172.19.150.82:2379 \  

```



```
13 --initial-cluster-token etcd-cluster \  
14 --initial-cluster infra1=https://172.19.104.231:2380,infra2=  
    https://172.19.104.230:2380,infra3=https  
    ://172.19.150.82:2380 \  
15 --initial-cluster-state new \  
16 --data-dir=/var/lib/etcd
```

启动节点 2:

```
1 /usr/local/bin/etcd \  
2 --name infra2 \  
3 --cert-file=/etc/kubernetes/ssl/kubernetes.pem \  
4 --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \  
5 --peer-cert-file=/etc/kubernetes/ssl/kubernetes.pem \  
6 --peer-key-file=/etc/kubernetes/ssl/kubernetes-key.pem \  
7 --trusted-ca-file=/etc/kubernetes/ssl/ca.pem \  
8 --peer-trusted-ca-file=/etc/kubernetes/ssl/ca.pem \  
9 --initial-advertise-peer-urls https://172.19.104.230:2380 \  
10 --listen-peer-urls https://172.19.104.230:2380 \  
11 --listen-client-urls http://127.0.0.1:2379 \  
12 --advertise-client-urls https://172.19.104.230:2379 \  
13 --initial-cluster-token etcd-cluster \  
14 --initial-cluster infra1=https://172.19.104.231:2380,infra2=  
    https://172.19.104.230:2380,infra3=https  
    ://172.19.150.82:2380 \  
15 --initial-cluster-state new \  
16 --data-dir=/var/lib/etcd
```

提示错误: error "remote error: tls: bad certificate", ServerName ""。输入如下命令检查 etcd 集群:

```
1 /usr/local/bin/etcdctl --endpoint https://127.0.0.1:2379 \  
2 --ca-file=/etc/kubernetes/ssl/ca.pem \
```

```
3 --cert-file=/etc/kubernetes/ssl/kubernetes.pem \
4 --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
5 cluster-health
6 # 查看集群单个节点状态
7 etcdctl endpoint status
8 # 查看集群多个节点状态
9 etcdctl endpoint status --cluster \
10 --cacert=/etc/kubernetes/ssl/ca.pem \
11 --cert=/etc/kubernetes/ssl/kubernetes.pem \
12 --key=/etc/kubernetes/ssl/kubernetes-key.pem \
13
14 etcdctl endpoint https://127.0.0.1:2379 \
15 --cacert=/etc/kubernetes/ssl/ca.pem \
16 --cert=/etc/kubernetes/ssl/kubernetes.pem \
17 --key=/etc/kubernetes/ssl/kubernetes-key.pem \
18
19 # etcd v3 查看集群状态
20 etcdctl endpoint health --cluster \
21 --cacert=/etc/kubernetes/ssl/ca.pem \
22 --cert=/etc/kubernetes/ssl/kubernetes.pem \
23 --key=/etc/kubernetes/ssl/kubernetes-key.pem
24 # v3 版本查看方式
25 etcdctl endpoint health
26 # 查看 etcd 集群的所有节点
27 etcdctl member list
28 etcdctl member add infra2 http://172.19.104.230:2380
```

健康的 etcd 集群使用如下命令可以检查：

```
1 [root@iZuf63refzweg1d9dh94t8Z ~]# etcdctl endpoint health
   -endpoints=localhost:2379
2 localhost:2379 is healthy: successfully committed proposal: took =
   3.656346ms
```

输入如下命令查看 etcd 集群信息：

```
1 # 查看 etcd 集群信息
2 curl http://localhost:2379/metrics
3 # 添加 etcd 节点
4 # 添加节点失败，提示错误 etcdserver: unhealthy cluster
5 # 删除集群数据 (/var/lib/etcd)，重新安装集群解决
6 etcdctl member add infra2 --peer-urls="http://172.19.104.230:2380"
7 etcdctl member add infra2 http://172.19.104.230:2380
```

v3 版本通过 REST Api 添加集群节点：

```
1 curl http://127.0.0.1:2379/v3beta/cluster/member/add -XPOST -H "
    Content-Type: application/json" -d '{"peerURLs": ["http
    ://172.19.104.230:2380"]}'
2
3 curl http://172.19.104.231:2379/health
```

查看集群健康状态：

```
1 etcdctl --endpoints=https://172.19.104.231:2379 \
2 --ca-file=/etc/kubernetes/ssl/kubernetes.pem \
3 --cert-file=/etc/kubernetes/pki/etcd/client.pem \
4 --key-file=/etc/kubernetes/pki/etcd/client-key.pem cluster-health
5
6 /opt/k8s/bin/etcdctl endpoint health --cluster --cacert=/etc/
    kubernetes/cert/ca.pem --cert=/etc/etcd/cert/etcd.pem --
    key=/etc/etcd/cert/etcd-key.pem
7
8 # 查看 etcd 集群健康状态
9 /opt/k8s/bin/etcdctl endpoint health --cluster
10
11 # 查看集群信息
12 curl -k --cert /etc/etcd/cert/etcd.pem --key /etc/etcd/cert/etcd-
```

```
key.pem https://172.19.150.82:2379/metrics
```

以表格形式查看集群信息：

```
1 /opt/k8s/bin/etcdctl endpoint status --write-out=table
```

etcd 部署完毕后，使用如下脚本查看集群的 Leader 信息<sup>31</sup>：

```
1 source /opt/k8s/bin/environment.sh
2 ETCDCTL_API=3 /opt/k8s/bin/etcdctl \
3   -w table --cacert=/etc/kubernetes/cert/ca.pem \
4   --cert=/etc/etcd/cert/etcd.pem \
5   --key=/etc/etcd/cert/etcd-key.pem \
6   --endpoints=${ETCD_ENDPOINTS} endpoint status
```

输出结果如图1.9所示，可以看到集群的 ID、集群数据库的大小、集群的版本、Term，Term 可以理解为周期（第几届、任期）的概念<sup>32</sup>，用 Term 作为一个周期，每个 Term 都是一个连续递增的编号，每一轮选举都是一个 Term 周期，在一个 Term 中只能产生一个 Leader；每次 Term 的递增都将发生新一轮的选举，Raft 保证一个 Term 只有一个 Leader，在 Raft 正常运转中所有的节点的 Term 都是一致的，如果节点不发生故障一个 Term（任期）会一直保持下去，当某节点收到的请求中 Term 比当前 Term 小时则拒绝该请求。

```
[root@iZuf63refzweg1d9dh94t8Z work]# ./check-etcd-cluster-leader.sh
```

ENDPOINT	ID	VERSION	DB SIZE	IS LEADER	RAFT TERM	RAFT INDEX
https://172.19.104.231:2379	5ab2d0e431f00a20	3.3.13	20 kB	true	5	16
https://172.19.104.230:2379	56298c42af788da7	3.3.13	20 kB	false	5	16
https://172.19.150.82:2379	84c70bf96ccff30f	3.3.13	20 kB	false	5	16

Figure 1.9: etcd 查看 Leader 节点

<sup>31</sup> 内容来源: <https://github.com/opsnull/follow-me-install-kubernetes-cluster/blob/master>

<sup>32</sup> 内容来源: <http://www.solinx.co/archives/415>

## 部署 master 节点

kubernetes master 节点包含的组件如下：

- kube-apiserver
- kube-scheduler
- kube-controller-manager

kube-scheduler、kube-controller-manager 和 kube-apiserver 三者的功能紧密相关。同时只能有一个 kube-scheduler、kube-controller-manager 进程处于工作状态，如果运行多个，则需要通过选举产生一个 leader；

## 下载最新版本的 Kubernetes 二进制文件

在中国大陆境内无法直接下载 Kubernetes 二进制文件，奇怪的是虽然使用了代理，但是还是无法通过 dl.k8s.io 域名进行下载，没有找到有价值的原因分析。所以此处采取的方式是在国外的主机上下载完毕后，拷贝文件到国内到云主机上进行部署（当然可以下载源码自己编译，不过估计要花不少时间和精力来处理其中遇到的问题），国外主机下载速度较快，在 50MB 每秒。注意以下命令在国外的服务器上执行，这里是用的是美国俄亥俄州的节点，当然香港、新加坡、台湾等物理距离较近的节点更好。

```
1 wget -c https://github.com/kubernetes/kubernetes/releases/download
    /v1.15.2/kubernetes.tar.gz
2 tar -xzvf kubernetes.tar.gz
3 cd kubernetes
4 ./cluster/get-kube-binaries.sh
```

从 Github 上下载下来的仅仅是一些脚本，这里下载的是 v1.15.2 版本的脚本，大概有 600KB 左右，完整的 Binary 文件估计有 1.5GB+。下载完毕后拷贝到阿里云服务器：

```
1 scp -i dolphin.pem -r ubuntu@ec2-23-53-88-011.us-east-2.compute.
    amazonaws.com:~/* .
```

将二进制文件拷贝到指定路径：

```
1 cp -r server/bin/{kube-apiserver,kube-controller-manager,kube-  
    scheduler,kubectl,kube-proxy,kubelet} /usr/local/bin/
```

### 配置和启动 kube-apiserver

kube-apiserver 是在部署 kubernetes 集群是最需要先启动的组件，也是和集群交互的核心组件。创建 kube-apiserver 的 service 配置文件 (路径: /usr/lib/systemd/system)，service 配置文件 kube-apiserver.service 内容：

```
1 [Unit]  
2 Description=Kubernetes API Service  
3 Documentation=https://github.com/GoogleCloudPlatform/kubernetes  
4 After=network.target  
5 After=etcd.service  
6  
7 [Service]  
8 EnvironmentFile=/etc/kubernetes/config  
9 EnvironmentFile=/etc/kubernetes/apiserver  
10 ExecStart=/usr/local/bin/kube-apiserver \  
11     $KUBE_LOGTOSTDERR \  
12     $KUBE_LOG_LEVEL \  
13     $KUBE_ETCD_SERVERS \  
14     $KUBE_API_ADDRESS \  
15     $KUBE_API_PORT \  
16     $KUBELET_PORT \  
17     $KUBE_ALLOW_PRIV \  
18     $KUBE_SERVICE_ADDRESSES \  
19     $KUBE_ADMISSION_CONTROL \  
20     $KUBE_API_ARGS  
21 Restart=on-failure  
22 Type=notify  
23 LimitNOFILE=65536  
24
```

```
25 [Install]
26 WantedBy=multi-user.target
```

/etc/kubernetes/config 文件的内容为：

```
1 #
2 # kubernetes system config
3 #
4 # The following values are used to
5 # configure various aspects of all
6 # kubernetes services, including
7 #
8 # kube-apiserver.service
9 # kube-controller-manager.service
10 # kube-scheduler.service
11 # kubelet.service
12 # kube-proxy.service
13 # logging to stderr means we get it in the systemd journal
14 KUBE_LOGTOSTDERR="--logtostderr=true"
15
16 # journal message level, 0 is debug
17 KUBE_LOG_LEVEL="--v=0"
18
19 # Should this cluster be allowed
20 # to run privileged docker containers
21 KUBE_ALLOW_PRIV="--allow-privileged=true"
22
23 # How the controller-manager, scheduler
24 # and proxy find the apiserver
25 KUBE_MASTER="--master=http://172.19.104.231:8080"
```

该配置文件同时被 kube-apiserver、kube-controller-manager、kube-scheduler、kubelet、kube-proxy 使用。添加 api server 配置文件/etc/kubernetes/apiserver，内容为：

```
1 KUBE_API_ADDRESS="--advertise-address=172.19.104.231 --bind-  
    address=172.19.104.231 --insecure-bind-address  
    =172.19.104.231"  
2 #  
3 # The port on the local server to listen on.  
4 #  
5 # Port minions listen on  
6 #  
7 # Comma separated list of nodes in the etcd cluster  
8 KUBE_ETCD_SERVERS="--etcd-servers=https://172.19.104.231:2379,  
    https://172.19.104.230:2379,https://172.19.150.82:2379"  
9 #  
10 # Address range to use for services  
11 KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"  
12 #  
13 # default admission control policies  
14 KUBE_ADMISSION_CONTROL="--admission-control=ServiceAccount,  
    NamespaceLifecycle,NamespaceExists,LimitRanger,  
    ResourceQuota"  
15 #  
16 # Add your own!  
17 KUBE_API_ARGS="--authorization-mode=RBAC --runtime-config=rbac.  
    authorization.k8s.io/v1beta1 --kubelet-https=true --enable  
    -bootstrap-token-auth --token-auth-file=/etc/kubernetes/  
    token.csv --service-node-port-range=30000-32767 --tls-cert  
    -file=/etc/kubernetes/ssl/kubernetes.pem --tls-private-key  
    -file=/etc/kubernetes/ssl/kubernetes-key.pem --client-ca-  
    file=/etc/kubernetes/ssl/ca.pem --service-account-key-file  
    =/etc/kubernetes/ssl/ca-key.pem --etcd-cafile=/etc/  
    kubernetes/ssl/ca.pem --etcd-certfile=/etc/kubernetes/ssl/  
    kubernetes.pem --etcd-keyfile=/etc/kubernetes/ssl/  
    kubernetes-key.pem --enable-swagger-ui=true --apiserver-
```



```
count=3 --audit-log-maxage=30 --audit-log-maxbackup=3 --  
audit-log-maxsize=100 --audit-log-path=/var/lib/audit.log  
--event-ttl=1h"
```

输入如下命令启动 kube-apiserver:

```
1 # 调整配置文件后执行此命令重新加载配置  
2 systemctl daemon-reload  
3 systemctl start kube-apiserver.service  
4 systemctl status kube-apiserver.service
```

启动 kube-apiserver 服务时提示 unknown flag: --experimental-bootstrap-token-auth, 原来 Bootstrap Token Authentication 在 1.9 版本已经变成了正式 feature(此次部署的版本是 v1.15), 将 kube-apiserver 配置文件/etc/kubernetes/apiserver 相应参数改为--enable-bootstrap-token-auth 即可。启动时提示错误:

```
1 clientconn.go:1251] grpc: addrConn.createTransport failed to  
connect to {172.19.150.82:2379 0 <nil>}. Err :connection  
error: desc = "transport: authentication handshake failed:  
x509: certificate signed by unknown authority (possibly  
because of \"crypto/rsa: verification error\" while trying  
to verify candidate authority certificate \"kubernetes\")  
\". Reconnecting...
```

将 pem 格式的证书转换为 crt 格式:

```
1 openssl x509 -outform der -in ca.pem -out ca.crt  
2 # 拷贝 CA 根证书  
3 cp /data/k8s/ssl/ca.crt /etc/pki/ca-trust/source/anchors/  
4 update-ca-trust
```

### 1.4.27 创建 kube-controller-manager 集群

创建证书签名请求:

```
1 cd /opt/k8s/work
2 cat > kube-controller-manager-csr.json <<EOF
3 {
4     "CN": "system:kube-controller-manager",
5     "key": {
6         "algo": "rsa",
7         "size": 2048
8     },
9     "hosts": [
10        "127.0.0.1",
11        "172.19.104.230",
12        "172.19.104.231",
13        "172.19.150.82"
14    ],
15     "names": [
16         {
17             "C": "CN",
18             "ST": "BeiJing",
19             "L": "BeiJing",
20             "O": "system:kube-controller-manager",
21             "OU": "4Paradigm"
22         }
23     ]
24 }
25 EOF
```

hosts 列表包含所有 kube-controller-manager 节点 IP。创建 kube-controller-manager 的 service 配置文件，文件路径/usr/lib/systemd/system/kube-controller-manager.service<sup>33</sup>。

```
1 [Unit]
2 Description=Kubernetes Controller Manager
```

<sup>33</sup>参考链接：<https://jimmysong.io/kubernetes-handbook/practice/master-installation.html>

```
3 Documentation=https://github.com/GoogleCloudPlatform/kubernetes
4
5 [Service]
6 EnvironmentFile=/etc/kubernetes/config
7 EnvironmentFile=/etc/kubernetes/controller-manager
8 ExecStart=/usr/local/bin/kube-controller-manager \
9     $KUBE_LOGTOSTDERR \
10    $KUBE_LOG_LEVEL \
11    $KUBE_MASTER \
12    $KUBE_CONTROLLER_MANAGER_ARGS
13 Restart=on-failure
14 LimitNOFILE=65536
15
16 [Install]
17 WantedBy=multi-user.target
```

配置文件/etc/kubernetes/controller-manager。

```
1 ###
2 # The following values are used to configure the kubernetes
3   controller-manager
4
5 # defaults from config and apiserver should be adequate
6
7 # Add your own!
8 KUBE_CONTROLLER_MANAGER_ARGS="--address=127.0.0.1 --service-
9   cluster-ip-range=10.254.0.0/16 --cluster-name=kubernetes
10  --cluster-signing-cert-file=/etc/kubernetes/ssl/ca.pem --
11  cluster-signing-key-file=/etc/kubernetes/ssl/ca-key.pem
12  --service-account-private-key-file=/etc/kubernetes/ssl/ca-
13  key.pem --root-ca-file=/etc/kubernetes/ssl/ca.pem --leader
14  -elect=true"
```

通过如下命令启动 controller-manager<sup>34</sup>:

```
1 systemctl start kube-controller-manager
```

无法通过服务启动时, 可直接通过命令行启动 Controller Manager:

```
1 nohup /usr/local/bin/kube-controller-manager --address=127.0.0.1
  --service-cluster-ip-range=10.254.0.0/16 --cluster-name=
  kubernetes --cluster-signing-cert-file=/etc/kubernetes/ssl
  /ca.pem --cluster-signing-key-file=/etc/kubernetes/ssl/ca-
  key.pem --service-account-private-key-file=/etc/
  kubernetes/ssl/ca-key.pem --root-ca-file=/etc/kubernetes/
  ssl/ca.pem --leader-elect=true --master=http:
  //172.19.104.231:8080 &
```

### 配置和启动 kube-scheduler

创建 kube-scheduler 的 service 配置文件, 文件路径/usr/lib/systemd/system/kube-scheduler.service。使用命令行启动 kubernetes scheduler:

```
1 /usr/local/bin/kube-scheduler --master=http://172.19.104.231:8080
  --leader-elect=true --address=127.0.0.1
```

### 验证 master 节点功能

master 所有组件启动完成后, 使用如下命令查看组件状态:

```
1 kubectl get componentstatuses
```

提示错误: The connection to the server localhost:8080 was refused - did you specify the right host or port? 此错误表示 kubectl 未正确配置。配置 kubeconfig 如下:

```
1 export KUBE_APISERVER="http://172.19.104.231:8080"
2 # 设置集群参数
```

<sup>34</sup>来源链接: <https://jimmysong.io/kubernetes-handbook/practice/master-installation.html>

```
3 kubectl config set-cluster kubernetes \  
4   --certificate-authority=/etc/kubernetes/ssl/ca.pem \  
5   --embed-certs=true \  
6   --server=${KUBE_APISERVER}  
7 # 设置客户端认证参数  
8 kubectl config set-credentials admin \  
9   --client-certificate=/etc/kubernetes/ssl/admin.pem \  
10  --embed-certs=true \  
11  --client-key=/etc/kubernetes/ssl/admin-key.pem  
12 # 设置上下文参数  
13 kubectl config set-context kubernetes \  
14   --cluster=kubernetes \  
15   --user=admin  
16 # 设置默认上下文  
17 kubectl config use-context kubernetes
```

添加部分基础配置后，连接拒绝的错误消失。输出各个组件的状态 (Component Status):

NAME	STATUS	MESSAGE	ERROR
scheduler	Healthy	ok	
controller-manager	Healthy	ok	
etcd-1	Healthy	{"health":"true"}	
etcd-0	Healthy	{"health":"true"}	
etcd-2	Healthy	{"health":"true"}	

### 安装 flannel 网络插件

覆盖网络 (Overlay Network) 就是应用层网络，它是面向应用层的，不考虑或很少考虑网络层，物理层的问题。详细说来，覆盖网络是指建立在另一个网络上的网络。该网络中的结点可以看作通过虚拟或逻辑链路而连接起来的。虽然在底层有很多条物理链路，但是这些虚拟或逻辑链路都与路径一一对应。例如：许多 P2P 网络就是覆盖网络，因为它运行在互连网的上层。覆盖网络允许对没有 IP 地址标识的目的主机路由信息，例如：Freenet 和 DHT（分布式哈希表）可以路由信息到一个存储特定文件的

结点，而这个结点的 IP 地址事先并不知道。覆盖网络被认为是一条用来改善互连网路由的途径，让二层网络在三层网络中传递，既解决了二层的缺点，又解决了三层的不灵活！Flannel 实质上是一种“覆盖网络 (overlay network)”，也就是将 TCP 数据包在另一种网络包里面进行路由转发和通信，目前已经支持 UDP、VxLAN、AWS VPC 和 GCE 路由等数据转发方式。数据从源容器中发出后，经由所在主机的 docker0 虚拟网卡转发到 flannel0 虚拟网卡，这是个 P2P 的虚拟网卡，flanneld 服务监听在网卡的另外一端。Flannel 通过 Etcd 服务维护了一张节点间的路由表，详细记录了各节点子网网段。源主机的 flanneld 服务将原本的数据内容 UDP 封装后根据自己的路由表投递给目的节点的 flanneld 服务，数据到达以后被解包，然后直接进入目的节点的 flannel0 虚拟网卡，然后被转发到目的主机的 docker0 虚拟网卡，最后就像本机容器通信一下的有 docker0 路由到达目标容器。所有的 node 节点都需要安装网络插件才能让所有的 Pod 加入到同一个局域网中，本文是安装 flannel 网络插件的参考文档。

### 下载分发 flannel 二进制文件

获取 flannel 部署文件<sup>35</sup>:

```
1 cd /opt/k8s/work
2 mkdir flannel
3 wget https://github.com/coreos/flannel/releases/download/v0.11.0/
   flannel-v0.11.0-linux-amd64.tar.gz
4 tar -xzvf flannel-v0.11.0-linux-amd64.tar.gz -C flannel
```

service 配置文件/usr/lib/systemd/system/flanneld.service。

```
1 [Unit]
2 Description=Flanneld overlay address etcd agent
3 After=network.target
4 After=network-online.target
5 Wants=network-online.target
6 After=etcd.service
7 Before=docker.service
8
```

<sup>35</sup>来源链接: <https://github.com/opsnull/follow-me-install-kubernetes-cluster/blob/master>

```
9 [Service]
10 Type=notify
11 EnvironmentFile=/etc/sysconfig/flanneld
12 EnvironmentFile=-/etc/sysconfig/docker-network
13 ExecStart=/usr/bin/flanneld-start \
14     -etcd-endpoints=${FLANNEL_ETCD_ENDPOINTS} \
15     -etcd-prefix=${FLANNEL_ETCD_PREFIX} \
16     $FLANNEL_OPTIONS
17 ExecStartPost=/usr/libexec/flannel/mk-docker-opts.sh -k
18     DOCKER_NETWORK_OPTIONS -d /run/flannel/docker
19 Restart=on-failure
20 [Install]
21 WantedBy=multi-user.target
22 RequiredBy=docker.service
```

/etc/sysconfig/flanneld 配置文件:

```
1 # Flanneld configuration options
2
3 # etcd url location. Point this to the server where etcd runs
4 FLANNEL_ETCD_ENDPOINTS="https://172.19.104.231:2379,https:
5     //172.19.104.230:2379,https://172.19.150.82:2379"
6
7 # etcd config key. This is the configuration key that flannel
8     queries
9
10 # For address range assignment
11 FLANNEL_ETCD_PREFIX="/kube-centos/network"
12
13 # Any additional options that you want to pass
14 FLANNEL_OPTIONS="-etcd-cafile=/etc/kubernetes/ssl/ca.pem -etcd-
15     certfile=/etc/kubernetes/ssl/kubernetes.pem -etcd-keyfile
16     =/etc/kubernetes/ssl/kubernetes-key.pem"
```

执行下面的命令为 docker 分配 IP 地址段。

```
1 etcdctl --endpoints=https://172.19.104.231:2379,https
   ://172.19.104.230:2379,https://172.19.150.82:2379 \
2   --ca-file=/etc/kubernetes/ssl/ca.pem \
3   --cert-file=/etc/kubernetes/ssl/kubernetes.pem \
4   --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
5   mkdir /kube-centos/network
6 etcdctl --endpoints=https://172.19.104.231:2379,https
   ://172.19.104.230:2379,https://172.19.150.82:2379 \
7   --ca-file=/etc/kubernetes/ssl/ca.pem \
8   --cert-file=/etc/kubernetes/ssl/kubernetes.pem \
9   --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
10  mk /kube-centos/network/config '{"Network":"172.30.0.0/16",
   SubnetLen":24,"Backend":{"Type":"vxlan"}}'
```

启动 flannel:

```
1 systemctl daemon-reload
2 systemctl enable flanneld
3 systemctl start flanneld
4 systemctl status flanneld
```

也可直接执行如下命令启动:

```
1 /usr/bin/flanneld -etcd-endpoints=https://172.19.104.231:2379,
   https://172.19.104.230:2379,https://172.19.150.82:2379 -
   etcd-prefix=/kube-centos/network -etcd-cafile=/etc/
   kubernetes/ssl/ca.pem -etcd-certfile=/etc/kubernetes/ssl/
   kubernetes.pem -etcd-keyfile=/etc/kubernetes/ssl/
   kubernetes-key.pem
2 /root/flanneld -etcd-endpoints=https://172.19.104.231:2379,https
```



```
://172.19.104.230:2379,https://172.19.150.82:2379 -etcd-
prefix=/kube-centos/network -etcd-cafile=/etc/kubernetes/
ssl/ca.pem -etcd-certfile=/etc/kubernetes/ssl/kubernetes.
pem -etcd-keyfile=/etc/kubernetes/ssl/kubernetes-key.pem
```

启动时提示错误: failed to retrieve network config: 100: Key not found (/kube-centos)。  
要先设置 key:

```
1 # v2 版本写法
2 etcdctl set /kube-centos/network/config '{ "Network":"10.1.0.0/16"
   }'
3 # v3 版本写法
4 etcdctl put /kube-centos/network/config '{ "Network":"10.1.0.0/16"
   }'
5 etcdctl get /kube-centos/network/config
6 # 查看所有的 key(etcd v3 写法)
7 etcdctl get / --prefix --keys-only
8 # 查看集群信息
9 ETCDCTL_API=3 etcdctl get /registry/pods --prefix -w json|python -
   m json.tool
```

实际操作过程中,即使设置好了 key,也还是会抛出 Key not found 的异常,初步确定是由于 flannel 截止到 2019 年 09 月 06 日,还没有兼容分布式数据库 etcd v3 版本,读取分布式数据库还是采用的 v2 版本的读取方式,而 v3 版本和 v2 版本的读取方式是不兼容的,所以会出现数据库明明已经设置了对应的 key,但是 flannel 还是始终无法读取到,提示 key 未找到错误。Etcd V2 和 V3 之间的数据结构完全不同,互不兼容,也就是说使用 V2 版本的 API 创建的数据只能使用 V2 的 API 访问,V3 的版本的 API 创建的数据只能使用 V3 的 API 访问。这就造成我们访问 etcd 中保存的 flannel 的数据需要使用 etcdctl 的 V2 版本的客户端,而访问 kubernetes 的数据需要设置 ETCDCTL\_API=3 环境变量来指定 V3 版本的 API。安装好 flannel 插件后,可以在 etcd 中检查是否有相应的数据,来判断是否运行正常。查询 etcd 中的内容可以看到:

```
1 /usr/local/bin/etcdctl --endpoints=${ETCD_ENDPOINTS} \
```

```
2 --cacert=/etc/kubernetes/ssl/ca.pem \  
3 --cert=/etc/kubernetes/ssl/kubernetes.pem \  
4 --key=/etc/kubernetes/ssl/kubernetes-key.pem \
```

注意不同版本的 `etcdctl` 其命令的参数有可能不一致，此处使用的 `etcdctl` 版本是 3.3.13。

### 安装完毕后验证

在各节点上部署 `flannel` 后，检查是否创建了 `flannel` 接口 (名称可能为 `flannel0`、`flannel.0`、`flannel.1` 等)：

```
1 source /opt/k8s/bin/environment.sh  
2 for node_ip in ${NODE_IPS[@]}  
3 do  
4     echo ">>> ${node_ip}"  
5     ssh ${node_ip} "/usr/sbin/ip addr show flannel.1|grep -w inet"  
6 done
```

输出的内容如下：

```
1 >>> 172.19.104.231  
2     inet 172.30.224.0/32 scope global flannel.1  
3 >>> 172.19.104.230  
4     inet 172.30.208.0/32 scope global flannel.1  
5 >>> 172.19.150.82  
6     inet 172.30.184.0/32 scope global flannel.1
```

在各节点上 `ping` 所有 `flannel` 接口 IP，确保能通：

```
1 source /opt/k8s/bin/environment.sh  
2 for node_ip in ${NODE_IPS[@]}  
3 do  
4     echo ">>> ${node_ip}"  
5     ssh ${node_ip} "ping -c 1 172.30.80.0"
```

```
6 ssh ${node_ip} "ping -c 1 172.30.32.0"
7 ssh ${node_ip} "ping -c 1 172.30.184.0"
8 done
```

此脚本会登陆各台主机，分别 ping 所有主机，保证所有主机之间能够互相 ping 通。在安装的过程中，遇到了本主机的 IP 可以 ping 通，但是其他主机的 IP 无法 ping 通的情况。添加 iptables 规则解决此问题 (实际上主机的 iptables 服务是没有启动的):

```
1 sudo iptables -P INPUT ACCEPT
2 sudo iptables -P OUTPUT ACCEPT
3 sudo iptables -P FORWARD ACCEPT
```

由于 82 这台主机使用的是 firewalld 防火墙，暂时还不清楚 flannel 服务需要开启哪个端口才可以正常访问，暂时开启所有端口来解决部署问题:

```
1 firewall-cmd --permanent --zone=public --add-port=0-65525/tcp
2 firewall-cmd --permanent --zone=public --add-port=0-65535/udp
3 firewall-cmd --reload
```

## 安装 Kubernetes Dashboard 插件

Kubernetes 的国外安装其实非常简单，国内安装的主要问题在于 kubernetes 部件所需的官方镜像在 <http://gcr.io>(Google Cloud Container Registry) 上，但是此网站在国内是无法访问的。当前安装 Kubernetes Dashboard 的时间是 2019 年 08 月 30 日，本文中安装的是 kubernetes dashboard v1.6.0<sup>36</sup>。执行如下命令部署 Dashboard Web UI(此方法不可取，在实际部署过程中，由于国内特殊的网络环境，部署所需要的部分镜像无法获取，在部署之前，还需要修改部分镜像的地址配置，将地址修改为国内网络可以访问的地址<sup>37</sup>，方可继续部署，否则后面需要花费大量的事件处理各种莫名的问题，浪费时间和精力):

<sup>36</sup>参考来源: <https://jimmysong.io/kubernetes-handbook/practice/dashboard-addon-installation.html>

<sup>37</sup>具体可以参考文章: <https://jimmysong.io/kubernetes-handbook/practice/dashboard-upgrade.html>

```
1 # 安装 Dashboard
2 kubectl apply -f https://raw.githubusercontent.com/kubernetes/
  dashboard/v1.10.1/src/deploy/recommended/kubernetes-
  dashboard.yaml
3 # 卸载 Dashboard
4 kubectl delete -f https://raw.githubusercontent.com/kubernetes/
  dashboard/v1.10.1/src/deploy/recommended/kubernetes-
  dashboard.yaml
```

部署完毕后，查看 Pods：

```
1 kubectl get pod --namespace=kube-system
2 # 查看所有命名空间的 Pods
3 kubectl get pods --all-namespaces
4 kubectl get pods,svc --all-namespaces -o wide
```

查看到镜像一直处于 Pending 状态，查看镜像的日志：

```
1 kubectl --namespace kube-system logs kubernetes-dashboard-7
  d75c474bb-b2lwd
```

如果日志没有输出，查看事件：

```
1 kubectl get events --namespace=kube-system
```

通过事件可以看出 kube-dns 和 metrics-server 服务 IP 范围指定与原有的 IP 范围不一致，提示 ClusterIPOutOfRange 错误。查看镜像的配置：

```
1 ./kubectl describe pod kubernetes-dashboard-7d75c474bb-b2lwd --
  namespace="kube-system"
```

启动 Scheduler 组件时，提示 Failed to update lock: Operation cannot be fulfilled on endpoints "kube-scheduler": StorageError: invalid object, Code: 4, Key: /registry/services/endpoints/kube-system/kube-scheduler, ResourceVersion: 0, AdditionalErrorMsg: Precondition failed: UID

in precondition: 121bc661-80f5-11e9-b3ce-00163e086f0c, UID in object meta: 49e84916-589a-4da5-b78a-761a1fe78285 错误。

查看集群信息:

```
1 # 查看集群 IP 信息
2 kubectl cluster-info dump | grep service-cluster-ip-range
3 kubectl get cm kubeadm-config -n kube-system -o yaml
```

### 通过 kubectl proxy 访问 dashboard

启动代理:

```
1 kubectl proxy
2 kubectl proxy --address='172.19.104.231' --port=8001 --accept-
  hosts='^*$'
```

输入如下链接通过 kubectl proxy 访问 Dashboard:

```
1 curl http://172.19.104.231:8001/api/v1/namespaces/kube-system/
  services/https:kubernetes-dashboard:/proxy/
2 http://kubernetes.example.com/api/v1/namespaces/kube-system/
  services/kube-ui/#/dashboard/
3 curl http://172.19.104.231:8001/api/v1/proxy/namespaces/kube-
  system/services/kubernetes-dashboard/
```

需要注意的是, Kubernetes Web UI 默认只支持在直接执行代理操作命令的机器上访问。查看 kube-system 命名空间下 pod 状态:

```
1 kubectl get pod --namespace=kube-system
```

### 安装 calico 插件

Calico 是一个纯三层的数据中心网络方案 (不需要 Overlay), 并且与 OpenStack、Kubernetes、AWS、GCE 等 IaaS 和容器平台都有良好的集成。Calico 在每一个计算节点利用 Linux Kernel 实现了一个高效的 vRouter 来负责数据转发, 而每个 vRouter 通过

BGP 协议负责把自己上运行的 workload 的路由信息像整个 Calico 网络内传播——小规模部署可以直接互联，大规模下可通过指定的 BGP route reflector 来完成。这样保证最终所有的 workload 之间的数据流量都是通过 IP 路由的方式完成互联的。Calico 节点组网可以直接利用数据中心的网络结构（无论是 L2 或者 L3），不需要额外的 NAT，隧道或者 Overlay Network。

#### 1.4.28 Tampermonkey 脚本收藏

##### Endless Google

在浏览 Google 搜索结果时，浏览到结尾时，需要点击下一页，才能够继续浏览下一页的内容，Endless Google 猴油插件可以在浏览到上一页末尾时，自动请求加载渲染下一页搜索结果，从而省去了手动点击下一页来加载后一页搜索结果这一步骤。这个脚本的作者是 tumpio，截至 2019 年 7 月 28 日，这款脚本可以使用。

#### 1.4.29 zsh 拷贝文件时通配符解析

在 zsh 下使用 scp 命令将远程服务器文件拷贝到本地：

```
1 scp -r app:/home/dolphin/model* .
```

会出现错误提示，zsh: no matches found: dabai-app:/home/dabai/model\*，这是由于 bash 和 zsh 在 glob<sup>38</sup>匹配时的不同行为导致<sup>39</sup>。如果 zsh 检查到命令中包含 globbing 表达式且没有匹配到内容时，会打印出 zsh: no matches found 提示，而 bash 的行为是，如果没有匹配到内容，则按照字面值返回，平时在 bash 下使用 scp 命令时，直接使用此命令拷贝是没有任何问题的，那是由于 bash 的规则是在 glob 匹配不到文件时，按照原始输入返回内容，这种行为机制隐藏掉了问题，例如/home/dabai/model\* 在 bash 下匹配不到对应的文件，那么 bash 就直接返回/home/dabai/model\* 字面量，返回的内容 scp 命令就可以解析执行了。但是在 zsh 下如果 glob 匹配不到会抛出错误（打印出未匹配到内容到提示语），此错误是本地 zsh 执行抛出，后续根本没有将内容传递给 scp 命令。可以在 bash 和 zsh 下分别执行如下命令：

```
1 echo globtest*
```

<sup>38</sup> 参考链接：[https://en.wikipedia.org/wiki/Glob\\_%28programming%29](https://en.wikipedia.org/wiki/Glob_%28programming%29)

<sup>39</sup> 参考链接：<https://www.cnblogs.com/timssd/p/7789119.html>

`bash` 下原样输出内容, `zsh` 下抛出匹配不到的异常(勉强能够理解,如果能够了解 `bash` 解析命令的过程就更好了)。解决此问题可以在 `*` 符号前加转义字符,就可以将路径当成普通文本。

```
1 scp -r app:/home/dolphin/model\* .
```

也可以用引号(单引号和双引号都可以,暂时还没有比较他们之间都区别。由单引号括起来的字符都作为普通字符出现。特殊字符用单引号括起来以后,会失去原有意义,而只作为普通字符解释。由双引号括起来的字符,除 `$`、倒引号(```)和反斜线(`\`)仍保留其特殊功能外,其余字符均作为普通字符对待。)将路径包含,当成普通字符串:

```
1 scp -r "app:/home/dolphin/model*" .
```

实际上,更标准的写法是将路径用引号包含的写法,只是平时由于 `bash` 的默认机制,这个问题没有暴露出来而已。

### 1.4.30 Git 慢

在使用 `Git` 做 `pull` 或者 `clone` 时,速度实在太慢,基本在 1024B 到 2KB 之间。虽说网速不快,但是油管 720P 视频还是可以播放的,不至于捉急到 2KB。`pull` 时提示信息如下:

```
1 bogon:the-books-of-mine dolphin$ git pull --verbose
2 POST git-upload-pack (972 bytes)
3 POST git-upload-pack (227 bytes)
4 remote: Enumerating objects: 22, done.
5 remote: Counting objects: 100% (22/22), done.
6 remote: Compressing objects: 100% (8/8), done.
7 error: RPC failed; curl 56 LibreSSL SSL_read: SSL_ERROR_SYSCALL,
   errno 54
8 fatal: the remote end hung up unexpectedly
9 fatal: early EOF
10 fatal: unpack-objects failed
```

本机恰好有代理，设置代理后，速度立即飙升到 300KB 左右：

```
1 git config --global http.proxy socks5://127.0.0.1:1080
2 git config --global https.proxy socks5://127.0.0.1:1080
```

原因一说是由于 Github 分发加速的域名 `assets-cdn.github.com` 遭到污染 (实际推测不仅仅是分发加速的域名遭到污染)，经过测试 (2019 年 7 月 24 测试，不保证不同的网络环境和运营商能够得到绝对一致的结果)，从测试结果来看此种推测成立。如果本地不加任何域名映射，直接用默认的域名解析，速度在 0 到 10KB 之间，添加 `global-ssl.fastly.Net` 和 `github.com` 域名映射 (域名指向到 IP 不保证永远不变)：

```
1 151.101.72.249 global-ssl.fastly.Net
2 192.30.253.112 github.com
```

此时速度能够达到 80KB 左右，添加分发加速 (CDN-Content Delivery Network 加速) 域名映射：

```
1 151.101.100.133 assets-cdn.github.com
```

此时速度在 100KB-250KB 之间，此时的速度对于大部分仓库应该就够用了，当然 3MB-30MB 的速度还是能够欣然接受的，怎么会嫌弃网速太快。有了这样的速度，再想想经过 DNS 污染的速度，还新增了不少幸福感。

### 1.4.31 排他锁更新记录

业务需要查询出数据后更新，采用 `select for upate` 方式加排他锁<sup>40</sup>：

```
1 SqlSession sqlSession = sqlSessionFactory.openSession(false);
2 Map<String, Object> queryParams = new HashMap<>();
3 RoomSeat roomSeatForUpdate = new RoomSeat();
4 try {
5     queryParams.put("roomTypeId", roomTypeId);
6     queryParams.put("roomPlayId", roomPlayId);
```

<sup>40</sup>参考链接：<https://www.cnblogs.com/xdp-gacl/p/4262895.html>



```
7      RoomSeat roomSeat = sqlSession.selectOne("
          selectRoomSeatForUpdate", queryParam);
8      if (roomSeat != null) {
9          roomSeatForUpdate.setStatus(1);
10         roomSeatForUpdate.setUserId(userId);
11         roomSeatForUpdate.setRobotFlag(robotFlag);
12         roomSeatForUpdate.setId(roomSeat.getId());
13         String statement = "com.sportswin.soa.room.dao.
            RoomSeatMapper.updateByPrimaryKeySelective";
14         sqlSession.update(statement, roomSeatForUpdate);
15     }
16     return roomSeat;
17 } catch (Exception e) {
18     log.error("匹配出错,参数: {},详细信息", JSON.toJSONString(
        roomSeatForUpdate), e);
19 } finally {
20     if (sqlSession == null) {
21         return null;
22     }
23     sqlSession.commit(true);
24     sqlSession.close();
25 }
```

InnoDB 行锁是通过给索引上的索引项加锁来实现的，这一点 MySQL 与 Oracle 不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB 这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB 才使用行级锁，否则，InnoDB 将使用表锁。

### 1.4.32 MyBatis 代码自动生成

配置文件：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE generatorConfiguration
```

```
3      PUBLIC "-//mybatis.org//DTD MyBatis Generator
      Configuration 1.0//EN"
4      "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
5
6  <generatorConfiguration>
7      <!-- classPathEntry:数据库的JDBC驱动,换成你自己的驱动位置 -->
8      <classPathEntry location="./mysql-connector-java-5.1.6-bin.jar"
9          "/>
10     <context id="MBG" targetRuntime="MyBatis3" defaultModelType="
11         conditional">
12         <property name="javaFileEncoding" value="UTF-8"/>
13         <property name="autoDelimitKeywords" value="true"/>
14         <property name="beginningDelimiter" value="'"/>
15         <property name="endingDelimiter" value="'"/>
16
17         <plugin type="org.mybatis.generator.plugins.KryoPlugin"/>
18         <plugin type="org.mybatis.generator.plugins.
19             SerializablePlugin"/>
20         <plugin type="org.mybatis.generator.plugins.
21             EqualsHashCodePlugin"/>
22         <plugin type="org.mybatis.generator.plugins.ToStringPlugin"
23             "/>
24         <plugin type="org.mybatis.generator.plugins.
25             PaginationPlugin"/>
26
27     <commentGenerator>
28         <!-- 去除自动生成的日期注释 -->
29         <property name="suppressDate" value="true"/>
30     </commentGenerator>
31
32     <jdbcConnection driverClass="com.mysql.jdbc.Driver"
33         connectionURL="jdbc:mysql://mysql.example.
34             com:3306/room_db?useUnicode=true&characterEncoding=UTF
```

```
28         -8"
                userId="root" password="password"></
jdbcTemplate>

29
30 <jdbcDriver>
31     <property name="forceBigDecimals" value="false"/>
32 </jdbcDriver>
33
34 <!-- targetProject:自动生成代码的位置 -->
35 <jdbcDriver targetPackage="com.sportswin.soa.room.
model.entity"
36         targetProject="../../../../soa-room-api/
src/main/java">
37     <property name="enableSubPackages" value="true"/>
38 </jdbcDriver>
39
40 <jdbcExampleGenerator targetPackage="com.sportswin.soa.
room.model.example"
41         targetProject="../../../../soa-room-api/
src/main/java">
42     <property name="enableSubPackages" value="true"/>
43 </jdbcExampleGenerator>
44
45 <sqlMapGenerator targetProject="../../../../soa-room-service/
src/main/resources/mybatis/mapper"
46         targetPackage="room">
47     <property name="enableSubPackages" value="true"/>
48 </sqlMapGenerator>
49
50 <jdbcClientGenerator type="XMLMAPPER" targetPackage="com.
sportswin.soa.room.dao"
51         targetProject="../../../../soa-room-
service/src/main/java">
```

```
52     <property name="enableSubPackages" value="true"/>
53 </javaClientGenerator>
54
55 <!--
56     tableName为对应的数据库表，
57     domainObjectName是要生成的实体类，
58     如果想要mapper配置文件加入sql的where条件查询
59     可以将enableCountByExample等设为true，
60     这样就会生成一个对应domainObjectName的Example类，
61     enableCountByExample等设为false时
62     就不会生成对应的Example类了。
63 -->
64
65 <table tableName="r_room_seat" domainObjectName="RoomSeat"
66     enableCountByExample="true"
67     enableUpdateByExample="true"
68     enableDeleteByExample="false"
69     enableSelectByExample="true"
70     selectByExampleQueryId="true">
71     <generatedKey column="ID" sqlStatement="MYSQL"
72     identity="true" />
73 </table>
74
75 </context>
</generatorConfiguration>
```

生成语句：

```
1 java -jar mybatis-generator-core-1.3.4.jar -configfile
    generatorConfig.xml -overwrite
```

### 1.4.33 参数校验

在 Controller 层接收 HTTP 请求参数时，往往离不开参数校验，校验的代码高度相似。当在微服务之间彼此调用时，需要校验的参数将更多。如果单纯手动编写校验代码。会造成许多重复的工作。Spring Validation 验证框架对参数的验证机制提供了 @Validated (Spring's JSR-303 规范，是标准 JSR-303 的一个变种)，javax 提供了 @Valid (标准 JSR-303 规范，JSR-303 是 Java EE 6 中的一项子规范，叫做 Bean Validation)。在检验 Controller 的入参是否符合规范时，使用 @Validated 或者 @Valid 在基本验证功能上没有太多区别。但是在分组、注解地方、嵌套验证等功能上两个有所不同。@Validated：用在方法入参上无法单独提供嵌套验证功能。不能用在成员属性（字段）上，也无法提示框架进行嵌套验证。@Valid：用在方法入参上无法单独提供嵌套验证功能。能够用在成员属性（字段）上，提示验证框架进行嵌套验证。

### 1.4.34 filebeat 递归读取文件

当开启 filebeat 程序的时候，它会启动一个或多个探测器（prospectors）去检测指定的日志目录或文件，对于探测器找出的每一个日志文件，filebeat 启动收割进程（harvester），每一个收割进程读取一个日志文件的新内容，并发送这些新的日志数据到处理程序（spooler），处理程序会集合这些事件，最后 filebeat 会发送集合的数据到指定的地点。filebeat 递归读取文件配置：

```
1 - type: log
2
3 # Change to true to enable this input configuration.
4 enabled: true
5
6 recursive_glob.enabled: true
7
8 # Paths that should be crawled and fetched. Glob based paths.
9 paths:
10   - /data/jenkins/soa-robot-service/*.log
11   - /data/logs/**/*.log
```

路径配置项/data/logs/\*\*/\*.log 会递归读取/data/logs 目录及其子目录下所有的 log 类型的文件，filebeat 的版本是 filebeat-7.2.0-linux-x86\_64。启用扩展 \*\* 到递归全局

模式。启用此功能后，\*\* 每个路径中的最右边将扩展为固定数量的全局模式。例如：  
/foo/\*\* 扩展到 /foo, /foo/\*, /foo/\*/\*, 等等。如果启用，则将单个扩展 \*\* 为 8 级深度 \*  
模式。此功能默认启用，设置 recursive\_glob.enabled 为 false 将其禁用。

### 1.4.35 Spring 手动获取 Bean

实现 ApplicationContextAware 接口：

```
1 @Component
2 public class SpringUtil implements ApplicationContextAware {
3
4     private static ApplicationContext applicationContext = null;
5
6     @Override
7     public void setApplicationContext(ApplicationContext
8         applicationContext) throws BeansException {
9         if (SpringUtil.applicationContext == null) {
10             SpringUtil.applicationContext = applicationContext;
11         }
12
13     public static ApplicationContext getApplicationContext() {
14         return applicationContext;
15     }
16
17     public static Object getBean(String name) {
18         return getApplicationContext().getBean(name);
19     }
20
21     public static <T> T getBean(Class<T> clazz) {
22         return getApplicationContext().getBean(clazz);
23     }
24 }
```

```
26 public static <T> T getBean(String name, Class<T> clazz) {  
27     return getApplicationContext().getBean(name, clazz);  
28 }  
29 }
```

手动获取 bean:

```
1 SessionUtil sessionUtil = SpringUtil.getBean(SessionUtil.class);
```

### 1.4.36 Jenkins 利用 Webhook 执行自动构建

人工执行代码构建比较繁琐, 重复劳动, 配置定时 pull 代码不够及时, 浪费资源。所以采用 Webhook 的方式。在 Gitlab 的项目设置 (Settings->Integrations, 注意不同的版本路径可能不同, 目前使用的版本是 GitLab Community Edition 11.11.0) 中添加 Jenkins 的触发构建回调地址。

### 1.4.37 Jackson 复杂类型反序列化

遇到 Jackson 在反序列化时, 返回的不是对应的 POJO 列表, 而是 LinkedHashMap, Jackson 在复杂类型 (泛型) 反序列化时, 默认反序列化为 LinkedHashMap。Jackson 处理一般的 JavaBean 和 Json 之间的转换只要使用 ObjectMapper 对象的 readValue 和 writeValueAsString 两个方法就能实现。但是如果要转换复杂类型 Collection 如 List<YourBean>, 那么就需要先反序列化复杂类型为泛型的 Collection Type。

```
1 private AppResponse getApp() throws Exception {  
2     // 获取到的类型为 LinkedHashMap 而不是 List<AppResponse>  
3     Response<List<AppResponse>> apps = appController.getApp();  
4     // 再次反序列化为实际的 POJO 列表  
5     List<AppResponse> appResponses = BeanConverter.  
        mapJsonToObjectList(new AppResponse(), JSON.toJSONString(  
            apps.getResult()), AppResponse.class);  
6     return appResponses.stream()  
7         .filter(item -> AppType.RED_ENVELOPE.name().equals(  
            item.getAppMark()))  
8         .findAny()
```

```
9         .orElse(null);  
10    }
```

### 1.4.38 Swagger 使用总结

**NumberFormatException: For input string: ""**

造成此问题的原因, 是 swagger 类 `AbstractSerializableParameter` 的方法 `getExample` 对 `example` 的判断策略, 新版本中如果 `example` 为空, 返回 `example`, 旧版本中返回为空。

```
1 <dependency>  
2   <groupId>io.swagger</groupId>  
3   <artifactId>swagger-annotations</artifactId>  
4   <version>1.5.22</version>  
5 </dependency>  
6 <dependency>  
7   <groupId>io.swagger</groupId>  
8   <artifactId>swagger-models</artifactId>  
9   <version>1.5.22</version>  
10</dependency>
```

另在发送 POST 请求时, 也可能出现参数转换异常, 如果在路径中传递参数, 在注解中要注解明白:

```
1 @ApiImplicitParams({  
2     @ApiImplicitParam(paramType = "path", name = "userId",  
3         required = true, value = "用户ID", dataType = "Long"),  
4 })
```

`paramType` 为 `path` 表示从 url 中获取参数, 对应注解中的 `PathVariable` 注解。`paramType` 表示参数放在哪个地方, 常用的有: `header`→ 请求参数的获取: `@RequestHeader`, `query`→ 请求参数的获取: `@RequestParam`, `path` (用于, `restful` 接口) → 请求参数的获取: `@PathVariable`, `body` (不常用), `form` (不常用)。



### Failed to start bean 'documentationPluginsBootstrapper'

启动 Zuul 项目时提示如下错误：

```
1 Failed to start bean 'documentationPluginsBootstrapper'; nested
   exception is com.google.common.util.concurrent.
   ExecutionError: java.lang.NoSuchMethodError: com.google.
   common.collect.FluentIterable.concat(Ljava/lang/Iterable;
   Ljava/lang/Iterable;)Lcom/google/common/collect/
   FluentIterable
```

造成该错误的原因是，springfox-swagger 2.9.2 中引用的 Guava 版本相对于项目中版本较高，此时和低版本的 Guava 引起冲突。springfox-swagger 2.9.2 中 Guava 的版本为 20。查看当前项目的 Guava 版本：

```
1 mvn dependency:tree|grep -C 10 guava
```

从输出可以知道目前引用的 Guava 版本是 19.0，在项目的 pom.xml 添加如下配置<sup>41</sup>：

```
1 <dependency>
2   <groupId>com.google.guava</groupId>
3   <artifactId>guava</artifactId>
4   <version>20.0</version>
5   <exclusions>
6     <exclusion>
7       <groupId>com.google.guava</groupId>
8       <artifactId>guava</artifactId>
9     </exclusion>
10  </exclusions>
11 </dependency>
```

<sup>41</sup> 参考链接：<https://github.com/springfox/springfox/issues/2616>

### 1.4.39 Zuul 统一集成 Swagger2 文档

项目开始的时候是将各个基础微服务统一以一个业务层的微服务暴露给各个客户端调用，后逐步扩展为多个业务逻辑微服务，UI 层调用端的请求从 Nginx 上映射到对应业务逻辑层微服务。随着基础微服务和业务逻辑层微服务不断增多，调用微服务变得越来越分散，同时前端要根据不同的项目定义不同的 URL 路径，目前调整起来成本越来越高。此时考虑引入微服务网关，将所有调用规范化，统一化管理，所有的请求通过网关来处理，前端同学只需要关注一个地方就可以，不需要操心后端的微服务如何架构，哪个模块该定义哪个 URL，调用哪个微服务，浏览哪个 Swagger 文档。引入了 Zuul 之后，需要将各个微服务的文档集成到一个地方，以下就是 Zuul 集成各个微服务系统文档几个步骤，启用 Zuul 的 swagger 文档<sup>42</sup>：

```
1 @Configuration
2 @EnableSwagger2
3 public class SwaggerConfig {
4     @Bean
5     public Docket createRestApi() {
6         return new Docket(DocumentationType.SWAGGER_2)
7             .apiInfo(apiInfo());
8     }
9
10    private ApiInfo apiInfo() {
11        return new ApiInfoBuilder()
12            .title("平台接口汇总")
13            .description("平台接口汇总-接口文档说明")
14            .termsOfServiceUrl("http://api.example.com")
15            .contact(new Contact("杨新华", "", "xinhuyang@example.com"))
16            .contact(new Contact("蒋小强", "", "xiaoqiang.jiang@example.com"))
17            .version("1.0.0")
18            .build();
19    }
```

<sup>42</sup>严重抄袭自：<https://www.jianshu.com/p/af4ff19afa04>

```
20 }
```

遍历微服务文档:

```
1 @Component
2 @Primary
3 public class DocumentationConfig implements
    SwaggerResourcesProvider {
4
5     private final RouteLocator routeLocator;
6
7     public DocumentationConfig(RouteLocator routeLocator) {
8         this.routeLocator = routeLocator;
9     }
10
11     @Override
12     public List<SwaggerResource> get() {
13         List resources = new ArrayList<>();
14         List<Route> routes = routeLocator.getRoutes();
15         routes.forEach(route -> {
16             String path = route.getFullPath().replace("**", "v2/
api-docs");
17             resources.add(swaggerResource(route.getId(), path, "
1.0.0"));
18         });
19         return resources;
20     }
21
22     private SwaggerResource swaggerResource(String name, String
location, String version) {
23         SwaggerResource swaggerResource = new SwaggerResource();
24         swaggerResource.setName(name);
25         swaggerResource.setLocation(location);
```

```
26     swaggerResource.setSwaggerVersion(version);
27     return swaggerResource;
28 }
29 }
```

定义映射:

```
1 @Configuration
2 public class ZuulConfig {
3     //自定义 serviceId 和路由之间的相互映射
4     @Bean
5     public PatternServiceRouteMapper serviceRouteMapper() {
6         return new PatternServiceRouteMapper(
7             "(?<project>^.+)-(?(subProject>.+)$)",
8             "${project}/${subProject}");
9     }
10 }
```

#### 1.4.40 Java 为什么要引入 Lambda 表达式

引入 Lambda 表达式可以带来如下好处, 第一个好处是可以利用多核 CPU。如果没有引入 Lambda 表达式, 一般处理集合的循环是采用外循环 (External Iterator) 写法:

```
1 for(int item: list) {
2     // 处理 item
3 }
```

引入 Lambda 表达式后, 集合循环可以采用内部循环 (Internal Iterator) 的写法:

```
1 list.forEach(s -> {
2     // 处理 item
3 })
```

第二个好处是提高代码的可读性。

### 1.4.41 Java 为什么要引入 InvokeDynamic 指令

Java 虚拟机为什么要引入 InvokeDynamic 指令？能够带来什么好处？要深入的理解，首先需要理解动态语言 (Dynamic Programming Language) 有什么好处。第一个好处是，动态语言拥有较高的生产力<sup>43</sup>。可以以更短的时间实现相同的功能 (享受好处的同时，也要明白动态语言在执行效率方面，通常情况下是与静态语言是存在差距的，所以在对执行效率要求很高的场合时，选择动态语言要仔细评估)。特别适合开发原型系统和一些工具。第二个好处是代码量少，动态语言通常可以以更少的代码量开发相同的功能，更少的代码通常意味着更少的 Bug。第三个好处是简单易学，动态语言语法相对简单，通常情况下学习成本较低。另外动态语言有一些细节的优势，比如静态语言中如果改变某个变量的类型，那么整个项目中所有定义变量的地方都要调整类型，而采用动态语言则不需要特意调整，比如 CSharp 里用 var 或者 dynamic 修饰的对象<sup>44</sup>。要具备这些好处，各种类型的虚拟机自然要想办法让动态类型的语言可以在自家平台上发挥优势，微软在 .NET 4.0 中新增加的 dynamic 关键字（它作用就是用来描述一个“无类型”的变量）以及相应的 DLR 支持，C# 4.0 便拥有了动态语言的特性<sup>45</sup>。Java 虚拟机则引入了 InvokeDynamic 关键字以及对应的特性，以便可以更好的支持动态类型语言运行在符合 Java 虚拟机规范的虚拟机上。Lambda 表达式作为在 Java 8 中实现匿名方法的一种途径而被引入，可以在某些场景中作为匿名类的替代方案。在字节码的层面上来看，Lambda 表达式被替换成了 invokedynamic 指令。

## 1.5 07 月

### 1.5.1 IntelliJ Idea 闪退

在使用 IntelliJ Idea 的过程中，经常遇到闪退的情况，一般情况下在用户的 Home 目录下会生成错误日志，但是此处也没有生成错误日志。微服务到目前为止有 9 个，而且 IntelliJ Idea 退出了，调试启动的程序还在后台持续运行，要将原来的所有进程 kill，再重新启动 IntelliJ Idea。比较糟糕的情况，一天会有 5 次，浪费了时间消耗了精力，简直让人崩溃。尝试开启 IntelliJ Idea 的内存指示。在配置菜单的 Appearance&Behavior 中开启 Show memory indicator 设置，开启了之后可以在 IDE 的右下角看到实时内存情况，默认配置的只有 750MB 左右，在微服务开启较多时，很有可能超出。PC 本

<sup>43</sup>参考链接: <https://www.infoq.cn/article/pros-and-cons-of-dynamic-languages>

<sup>44</sup>参考链接: <https://docs.microsoft.com/zh-cn/dotnet/framework/reflection-and-codedom/dynamic-language-runtime-overview>

<sup>45</sup>参考链接: <https://www.infoq.cn/article/jdk-dynamically-typed-language>

身的内存有 16GB，调整为 3GB 后暂时未遇到闪退的情况 (还是会有闪退情况，次数明显减少，配合 shell 脚本结束进程，引入 Run Dashboard 管理微服务，情况有所改善)。IntelliJ Idea 占用内存大小的配置文件在 Mac OS 下存放的目录路径为：/Users/dolphin/Library/Application Support/JetBrains/Toolbox/apps/IDEA-U/ch-0/191.7479.19。这个是目前配置：

```
1 -Xmx3756m
2 -XX:ReservedCodeCacheSize=240m
3 -XX:+UseCompressedOops
4 -Dfile.encoding=UTF-8
5 -XX:+UseConcMarkSweepGC
6 -XX:SoftRefLRUPolicyMSPerMB=50
7 -ea
8 -Dsun.io.useCanonCaches=false
9 -Djava.net.preferIPv4Stack=true
10 -Djdk.http.auth.tunneling.disabledSchemes=""
11 -XX:+HeapDumpOnOutOfMemoryError
12 -XX:-OmitStackTraceInFastThrow
13 -Xverify:none
14
15 -XX:ErrorFile=$USER_HOME/java_error_in_idea_%p.log
16 -XX:HeapDumpPath=$USER_HOME/java_error_in_idea.hprof
17 -Dide.no.platform.update=true
```

### 1.5.2 Bash 使用 Map 数据结构

Shell 脚本使用 Map 数据结构来保存端口和应用对名字，循环结束对应对进程：

```
1 #!/usr/local/bin/bash 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 # 定义应用的端口和名称
13 declare -A map=(
14     ["8761"]="Eureka服务"
15     ["11001"]="用户微服务"
16     ["11002"]="钱包微服务")
17
18 for key in ${!map[@]}
19 do
20     read -p "是否要结束${map[$key]}? Please input (Y/N) : " yn
21     if [ "$yn" == "Y" ] || [ "$yn" == "y" ]; then
22         PID_COUNT=$(lsof -t -i:${key}|wc -l)
23         if [[ ${PID_COUNT} -gt 1 ]]; then
24             kill -9 $(lsof -t -i:${key})
25         else
26             echo "Process not found"
27         fi
28     elif [ "$yn" == "N" ] || [ "$yn" == "n" ]; then
29         echo "No!"
30     else
31         echo "Input Error"
32     fi
33 done
```

Bash 映射 (map) 在文档里叫做关联数组 (associated array), 使用关联数组的最低 Bash 版本是 4.1.2(Bash 4.0 在 2009 年 2 月份发布, Bash 5.0 在 2019 年 1 月份发布)。由于我的默认 Bash 是 oh my zsh, 在使用 zsh 命令执行时, 始终提示无法执行二进制文件 (cannot execute binary file)。所以执行脚本时需要指定 Bash 4 的绝对路径, 使用如

下命令来执行：

```
1 /usr/local/bin/bash ./batch-terminal-process.sh
```

### 1.5.3 macOS Mojave 升级 Bash 到 4.0 版本

截止到 2019 年 7 月 14 日，macOS Mojave 默认用的还是 3.x 版本，可能是 License 原因<sup>46</sup>，也许是由于 Bash 4 到 License 做了某些调整，对苹果不利，所以没有升级，谁知道呢。我的 Mac Book Pro 用的 bash 版本是 GNU bash, version 3.2.57(1)-release (x86\_64-apple-darwin18)。bash 3.x 版本不支持 map 数据结构，而在脚本中却需要使用 map 结构来简化代码，所以需要将 bash 升级到 4.x 版本。升级 bash 运行如下指令：

```
1 brew install bash
2 sudo bash -c 'echo /usr/local/bin/bash >> /etc/shells'
3 chsh -s /usr/local/bin/bash
```

安装好后，重新启动终端即可生效。

### 1.5.4 Websocket 常见问题处理

Websocket 经常自动关闭连接，提示错误：

```
1 {
2     "closeCode": "CLOSED_ABNORMALLY",
3     "reasonPhrase": "Unexpected Status of SSLEngineResult after an
        unwrap() operation"
4 }
```

从 `WsRemoteEndpointImplBase` 类的 `sendMessageBlock` 方法的 313 行抛出。`WsRemoteEndpointImplClient` 调用底层 `doWrite` 方法，实现实际向服务端写入的逻辑，在 `UnixAsynchronousSocketChannelImpl` 类的 `write` 方法中。1006 `CLOSE_ABNORMAL` 用于期望收到状态码时连接非正常关闭 (也就是说，没有发送关闭帧)。如果 `WebSocket` 连接已经关闭，且端点没有接收到 `Close` 状态码 (例如可能发生在底层传输连接丢失

<sup>46</sup>参考来源：<https://akrabat.com/upgrading-to-bash-4-on-macos/>



时), WebSocket 连接 Close Code 被认为是 1006。此问题初步确定是由于没有发送心跳, 服务端主动断开连接导致, 新增客户端心跳发送代码:

```
1 @OnOpen
2 public void onOpen(Session userSession) {
3     executorService.scheduleAtFixedRate(new Runnable() {
4         @Override
5         public void run() {
6             String data = "Ping";
7             ByteBuffer payload = ByteBuffer.wrap(data.getBytes());
8             if (userSession != null) {
9                 userSession.getBasicRemote().sendPing(payload);
10                log.info("发送心跳");
11            }
12        }
13    }, 5, 10, TimeUnit.SECONDS);
14 }
```

5 表示 initialDelay, initialDelay 是说系统启动后, 需要等待多久才开始执行。10 表示 period, period 为固定周期时间, 按照一定频率来重复执行任务。Websocket 关闭时提示:

```
1 closing websocket,reason:{"closeCode":"T00_BIG","reasonPhrase":"
    The decoded text message was too big for the output buffer
    and the endpoint does not support partial messages"}
```

错误编号 1009 CLOSE\_TOO\_LARGE 由于收到过大的数据帧而断开连接. 设置 BufferSize:

```
1 public WebsocketClientEndpoint(URI endpointURI) {
2     WebSocketContainer container = ContainerProvider.
        getWebSocketContainer();
3     container.setDefaultMaxTextMessageBufferSize(10000);
4     container.setDefaultMaxBinaryMessageBufferSize(50000);
```

```
5     container.connectToServer(this, endpointURI);  
6 }
```

### 1.5.5 Shell 常用脚本

根据进程的名称判断进程是否存在，若存在则结束进程：

```
1 PID='ps -ef|grep -w ${PROGRAM_NAME}|grep -v grep|cut -c 9-15'  
2 if [[ ${PID} -gt 1 ]]; then  
3     kill -15 ${PID}  
4 else  
5     echo "Process not found"  
6 fi
```

另一种方式是直接使用 `pgrep` 命令判断进程是否存在，更加简洁：

```
1 count=$(pgrep ${APP_NAME} | wc -l)  
2 if [[ ${PID} -gt 1 ]]; then  
3     kill -15 ${PID}  
4 else  
5     echo "Process not found"  
6 fi
```

### 1.5.6 容器编排 (Container Orchestration) 方式管理 Docker

### 1.5.7 Docker 方式部署应用

Docker 方式部署主要包含项目代码和 Dockerfile 提交，Jenkins 持续集成工具编译构建 (Compile&Build) 项目，构建完毕后 Docker 打包，推送到 Docker 镜像到远端镜像服务器 (Harbor<sup>47</sup>)，最后 Kubelets 等容器管理平台从镜像服务器 pull 镜像运行应用。Docker 打包镜像定义的 Dockerfile 代码如下：

---

<sup>47</sup>Harbor 是一个原生云计算基金会项目，它提供了一个原生云注册中心，用于容器镜像存储、签名和扫描。

```
1 FROM centos
2
3 MAINTAINER jiangxiaoqiang (jiangtingqiang@gmail.com)
4
5 ENV JAVA_VERSION 8u191
6 ENV BUILD_VERSION b12
7
8 RUN yum -y install wget
9
10 RUN wget -c --header "Cookie: oraclelicense=accept-securebackup-
    cookie" http://download.oracle.com/otn-pub/java/jdk/8u131-
    b11/d54c1d3a095b4ff2b6607d096fa80163/jdk-8u131-linux-x64.
    rpm -O /tmp/jdk-8-linux-x64.rpm
11
12 RUN yum -y install /tmp/jdk-8-linux-x64.rpm
13
14 ADD soa-room-service-1.0.0-SNAPSHOT.jar /soa-room-service-1.0.0-
    SNAPSHOT.jar
15
16 ADD start-docker.sh /start-docker.sh
17
18 EXPOSE 13003
19
20 ENTRYPOINT /start-docker.sh
```

Dockerfile 第一条必须为 FROM 指令, FROM 命令是 Dockerfile 中唯一不可缺少的命令, 它为最终构建出的镜像设定了一个基础镜像 (base image)。如果同一个 Dockerfile 创建多个镜像时, 可使用多个 FROM 指令, 每个镜像一次。Run 指令, 格式为 Run 或者 Run ["executable", "Param1", "param2"]。前者在 shell 终端上运行, 即/bin/sh -C, 后者使用 exec 运行。例如: RUN ["/bin/bash", "-c", "echo hello"]。每条 run 指令在当前基础镜像执行, 并且提交新镜像。当命令比较长时, 可以使用“/”换行。使用如下命令生成镜像:

```
1 # 构建镜像, 镜像名称为 soa-room-service
2 docker build -t="soa-room-service" .
```

启动本地镜像:

```
1 # 启动本地镜像
2 # 第一个 soa-room-service 为实例名称
3 # 第二个 soa-room-service 为镜像名称
4 docker run -p 9081:9081 --name soa-room-service soa-room-service
```

如果在执行 `docker run` 时没有指定 `--name`, 那么 `daemon` 会自动生成一个随机字符串 `UUID`。但是对于一个容器来说有个 `name` 会非常方便, 当你需要连接其它容器时或者类似需要区分其它容器时, 使用容器名称可以简化操作。无论容器运行在前台或者后台, 这个名字都是有效的。一般情况下, 需要将镜像推送到公网仓库 (或者自建仓库, 但是要保证生产环境容器管理平台可以 `pull` 自建仓库中的容器), 容器管理平台 `pull` 容器镜像进行部署。此处暂时使用阿里云提供的容器管理中心 (Aliyun Docker Hub)。登录阿里云管理控制台, 在菜单中找到容器镜像服务, 在界面上创建对应的镜像仓库。如下脚本将构建镜像推送到阿里云:

```
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 # 也可以使用命令 set -o nounset
5 set -u
6
7 # 当任何一行命令执行失败时, 自动退出脚本
8 # 也可以使用命令 set -o errexit
9 set -e
10
11 set -x
12
13 /usr/bin/expect <<-EOF
14 set timeout 30
```

```
15 # 镜像推送到阿里云
16 # spawn 命令激活一个 Unix 程序来进行交互式的运行
17 spawn sudo docker login --username=dolphin registry.cn-shanghai.
    aliyuncs.com
18 # expect 命令等待进程的某些字符串。
19 # expect 支持正规表达式并能同时等待多个字符串
20 # 并对每一个字符串执行不同的操作
21 # expect 还能理解一些特殊情况，如超时和遇到文件尾
22 expect "password:"
23 # send 命令向进程发送字符串
24 send "123456\r"
25 expect eof
26 EOF
27
28 # 镜像推送到阿里云
29 sudo docker tag dolphin-service registry.cn-shanghai.aliyuncs.com/
    app/dolphin:latest
30 sudo docker push registry.cn-shanghai.aliyuncs.com/app/dolphin:
    latest
```

推送脚本在阿里云容器管理界面有详细文档，直接拷贝调整必要参数即可。唯一要解决的问题是，采用自动化脚本部署，阿里云在推送镜像到远程仓库时需要先登录，登录需要在交互模式下输入密码，如果是手动执行 shell 脚本，执行脚本在终端中按照提示输入对应密码即可，而 shell 命令默认是不支持后台交互的，这里用到了 expect<sup>48</sup>来自动输入密码。Expect 的工作方式象一个通用化的 Chat 脚本工具。Chat 脚本最早用于 UUCP 网络内，以用来实现计算机之间需要建立连接时进行特定的登录会话的自动化。Chat 脚本由一系列 expect-send 对组成：expect 等待输出中输出特定的字符，通常是一个提示符，然后发送特定的响应。

### 1.5.8 EFK Docker 部署

Docker Engine 在安装时自动创建 bridge 网络。该网络对应于 Engine 传统依赖的 docker0 网桥。除该网络外，也可创建自己的 bridge 或 overlay 网络。bridge 网络驻留

<sup>48</sup>参考链接：<https://en.wikipedia.org/wiki/Expect>

在运行 Docker Engine 实例的单个主机上。overlay 网络可跨越运行 Docker Engine 的多个主机。如果您运行 `docker network create` 并仅提供网络名称，它将为您创建一个桥接网络。使用如下命令创建一个桥接网络：

```
1 docker network create esnet
```

安装 Docker compose：

```
1 sudo curl -L "https://github.com/docker/compose/releases/download
  /1.24.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/
  local/bin/docker-compose
```

安装 ElasticSearch、Fluentd、Kibana(截止 2019 年 07 月 08 日)：

```
1 docker pull docker.elastic.co/elasticsearch/elasticsearch:7.2.0
```

### 1.5.9 集成 Request ID(全链路监测)

在分布式系统中，一个请求经过多个系统，最终完成处理返回给用户。传统处理问题的方式是登陆各个系统主机，排查日志，近期上线了 EFK，此时日志可以集中查看，但是无法方便的识别通一个请求的一条调用链路，如果能将 Request 唯一编号集成在日志中，那么就可以在一处排查链路哪一个节点发生故障。在第一个节点的请求拦截器中生成唯一 ID：

```
1 public void apply(RequestTemplate tpl) {
2     tpl.header("token", SessionUtil.getAuthToken());
3     tpl.header("requestId", UUID.randomUUID().toString());
4 }
```

借助 logback MDC 机制，MDC 为“Mapped Diagnostic Context”(映射诊断上下文)，将 Request ID 通过 logback 打印出来：

```
1 MDC.put("requestId", requestId);
```

logback.xml 中配置日志输出带上 Request ID：

```
1 <appender name="file" class="ch.qos.logback.core.rolling.  
    RollingFileAppender">  
2 <file>${log.path}</file>  
3 <rollingPolicy class="ch.qos.logback.core.rolling.  
    TimeBasedRollingPolicy">  
4 <fileNamePattern>${log.path}.%d{yyyy-MM-dd}.zip</  
    fileNamePattern>  
5 </rollingPolicy>  
6 <encoder>  
7 <pattern>%date %level %X{requestId} [%thread] %logger{36}  
    [%file : %line] %msg%n  
8 </pattern>  
9 </encoder>  
10 </appender>
```

更好的方案可以集成全链路监控系统，市面上的全链路监控理论模型大多都是借鉴 Google Dapper 论文。Zipkin：由 Twitter 公司开源，开放源代码分布式的跟踪系统，用于收集服务的定时数据，以解决微服务架构中的延迟问题，包括：数据的收集、存储、查找和展现。Pinpoint：一款对 Java 编写的大规模分布式系统的 APM 工具，由韩国人开源的分布式跟踪组件。Skywalking：国产的优秀 APM 组件，是一个对 JAVA 分布式应用程序集群的业务运行情况进行追踪、告警和分析的系统。

### 1.5.10 Feign 传递公共请求头

在微服务间使用 Feign 互相调用时，系统做了公共的认证，需要在请求头中传入登陆获取到的 Token。由于是公共 Token，在 Feign 中的参数和某个方法上定义接收 Token 是不合适的，否则每个方法都需要添加。通过定义一个公共的拦截器来放入 Token：

```
1 @Component  
2 public class FeignBasicAuthRequestInterceptor implements  
    RequestInterceptor {  
3     public FeignBasicAuthRequestInterceptor() {
```

```
4
5     }
6
7     @Override
8     public void apply(RequestTemplate template) {
9         template.header("token", System.getProperty("dolphin.auth.
10            token"));
11     }
12 }
```

系统配置中注入拦截器:

```
1 @Configuration
2 public class FeignConfiguration {
3     /**
4      * Log level
5      * @return
6      */
7     @Bean
8     Logger.Level feignLoggerLevel() {
9         return Logger.Level.FULL;
10    }
11
12    /**
13     Create a Feign request interceptor
14     set the authentication token before sending the request
15     and each micro service sets the token to the environment
16     variable to achieve universal
17     * @return
18     */
19    @Bean
20    public FeignBasicAuthRequestInterceptor
21        basicAuthRequestInterceptor() {
```



```
20     return new FeignBasicAuthRequestInterceptor();  
21 }  
22 }
```

同时在合适的地方设置 `dolphin.auth.token` 的值即可<sup>49</sup>。

### 1.5.11 IntelliJ Idea 编译找不到符号

在使用 IntelliJ Idea 时经常会遇到上一次编译没有任何问题，下一次编译就会提示找不到符号 (Can not find symbol)，代码肯定是没有问题的。此时可以采取如下方案来解决。

- 首先，确定代码是否真的包含相应的字段和方法 (一般都不是代码问题)
- 其次确定 IntelliJ Idea 是否开启了注解 (annotation)，在菜单 Preference -> Build, Execution, Deployment -> Compiler -> Annotation Processors，勾选上 Enable annotation processing 选项
- 如果确定代码已经添加相应的字段，同时也开启了系统设置中的注解。可以尝试刷新 Maven 后再行编译
- 如果提示依旧，可以尝试重启 IntelliJ Idea，点击菜单 File > Invalidate Caches/Restart 进行重启
- 如果提示依旧，可以尝试重新导入 Maven/Gradle 项目，注意选择删除现有的重新导入
- 如果提示依旧，可以尝试先在命令行下编译项目 (遇到一次，手工使用命令行编译成功后，IntelliJ Idea 自动修复)，再在 IntelliJ 中启动项目
- 如果提示依旧，可以右键项目重新构建 (Rebuild)<sup>50</sup>

### 1.5.12 微服务常见问题

#### The target server failed to respond

在本机请求用户微服务时，提示错误：I/O exception caught when processing request to ->http://192.168.2.80:11001: The target server failed to respond。尝试请求本机其他微服务，也提示同样的错误。首先怀疑是本机代理问题，关闭本机代理后问题消失，Java socket 默认使用系统代理。

<sup>49</sup>本文严重抄袭自：<http://www.programmersought.com/article/4993606187/>

<sup>50</sup>参考来源：<https://stackoverflow.com/questions/12132003/getting-cannot-find-symbol-in-java-project-in-intellij>

**feign.RetryableException: Read timed out executing GET**

在使用 openfeign<sup>51</sup>(10.1.0) 调用服务时 (2019 年 06 月 26 日), 出现错误 Read timed out executing GET, 原因是 feign 调用超时, 调整如下配置即可:

```
1 feign.client.config.default.connectTimeout: 16000
2 feign.client.config.default.readTimeout: 16000
```

但是到 openfeign 源码里面查看, feign client 默认的 connectTimeout 为 10s, readTimeout 为 60s, 不知何故。如果是 hystrix 调用请求超时, 则会提示类似错误: GetAPI-Command timed-out and no fallback available。此时解决可以暂时关闭熔断机制, 更好的做法是为每个接口指定 fallback 方法, 处理此类异常。

**Load balancer does not have available server for client: dolphin-envelope**

在使用 feign 调用其他微服务客户端时, 提示错误 Load balancer does not have available server for client:dolphin-envelop。经过排查, 是在 feign 接口层定义微服务名称时, 将名字拼写错误。在 application.properties 中指定的名称是 dolphin-envelope:

```
1 spring.application.name=dolphin-envelope
```

而在 Feign 中注解的名字少拼写了一个 e:

```
1 @FeignClient(name = "soa-red-envelop-service")
```

调整即可修复此问题。

### 1.5.13 IntelliJ Idea 循环依赖

IntelliJ Idea 循环依赖 Annotation processing is not supported for module cycles。错误现象日志: Error:java: Annotation processing is not supported for module cycles. Please ensure that all modules from cycle [soa-misc,soa-api] are excluded from annotation processing。通过 Analyze->Analyze Module Dependencies 发现的确是循环依赖。

---

<sup>51</sup>OpenFeign 是 Netflix 开源的参照 Retrofit, JAXRS-2.0, and WebSocket 的一个 http client 客户端, 致力于减少 http client 客户端构建的复杂性。

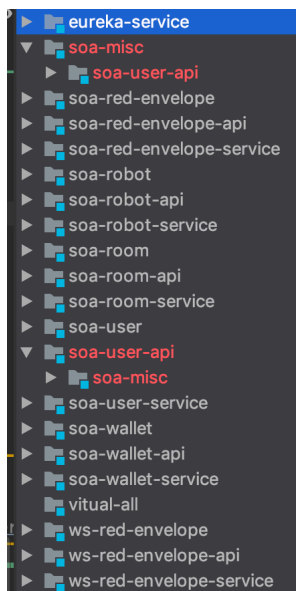


Figure 1.10: IntelliJ Idea 循环引用

### 1.5.14 IntelliJ Idea 热部署

在开发的过程中，重启 App 使代码修改生效花的时间比较长。研究了下热部署方案，JRebel 一年一个 License 是 550 美金，在 2019 年 6 月左右的汇率下，约合人民币 3300 元，价格对本人来说还是有点小贵（巨贵好吗），而且本人不在走投无路的情况下，拒绝使用破解版。恰好有一个开源的 `spring-boot-devtools` 的模块可以一试，Spring 为开发者提供了一个名为 `spring-boot-devtools` 的模块来使 Spring Boot 应用支持热部署，提高开发者的开发效率，避免修改代码后手动重启 Spring Boot 应用。`devtools` 深层原理是使用了两个 `ClassLoader`，一个 `ClassLoader` 加载那些不会改变的类（第三方 Jar 包），另一个 `ClassLoader` 加载会更改的类，称为 `restart ClassLoader`，在有代码更改的时候，原来的 `restart ClassLoader` 被丢弃，重新创建一个 `restart ClassLoader`，由于需要加载的类相比较少，所以实现了较快的重启速度，节省了时间。第一步、先设置我们的 `pom.xml` 文件，加入依赖，首先是把下面代码在 `<dependencies>` 中：

```
1 <!--添加热部署-->
2 <dependency>
3     <groupId>org.springframework.boot</groupId>
```

```
4     <artifactId>spring-boot-devtools</artifactId>
5     <optional>true</optional>
6     <scope>true</scope>
7 </dependency>
```

另外下面的代码是放在 <build> 下面 <plugins> 里：

```
1 <plugin>
2     <!--热部署配置-->
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-maven-plugin</artifactId>
5     <configuration>
6         <!--fork:如果没有该项配置,整个devtools不会起作用-->
7         <fork>true</fork>
8     </configuration>
9 </plugin>
```

第二步、设置 IDEA 的自动编译：在 IntelliJ Idea 的 File-Settings-Compiler 中勾选 Build Project automatically。Mac 下快捷键 Option + Command + Shift + /, 选择 Registry, 勾上 Compiler autoMake allow when app running 即可。到这里，所有到配置完毕，修改代码按保存后，会触发自动重新 Load 类，不需要再手动重启。在 IntelliJ Idea 的配置窗口配置系统更新策略后，在 Services(部分版本 IntelliJ Idea 的 Run Dashboard 面板)面板上会出现刷新按钮，如图1.11所示：

### 1.5.15 JVM 内存模型 (Java Memory Model)

Java 内存模型的相关知识在 JSR-133: Java Memory Model and Thread Specification 中描述的。JMM 是和多线程相关的，他描述了一组规则或规范，这个规范定义了一个线程对共享变量的写入时对另一个线程是可见的。Java 内存模型 (Java Memory Model, JMM) 就是一种符合内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，保证了 Java 程序在各种平台下对内存的访问都能得到一致效果的机制及规范。目的是解决由于多线程通过共享内存进行通信时，存在的原子性 (Atomic)、可见性 (缓存一致性) 以及有序性问题。

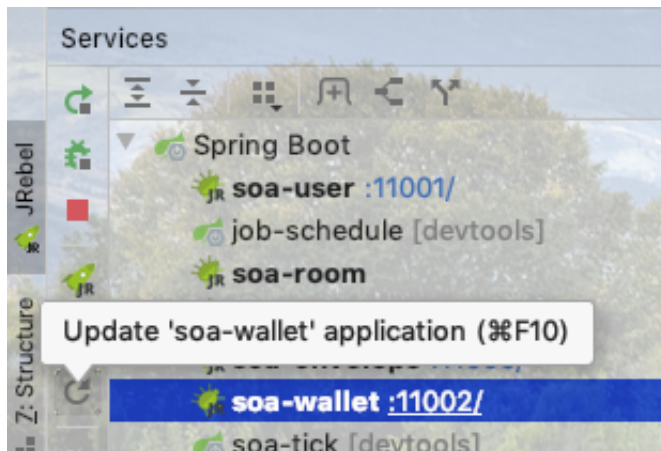


Figure 1.11: 手动热加载按钮

### 原子性

在 Java 中，为了保证原子性，提供了两个高级的字节码指令 `monitorenter` 和 `monitorexit`。在 `synchronized` 的实现原理文章中，介绍过，这两个字节码，在 Java 中对应的关键字就是 `synchronized`。

因此，在 Java 中可以使用 `synchronized` 来保证方法和代码块内的操作是原子性的。

### 可见性

Java 内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值的这种依赖主内存作为传递媒介的方式来实现的。

Java 中的 `volatile` 关键字提供了一个功能，那就是被其修饰的变量在被修改后可以立即同步到主内存，被其修饰的变量在每次是用之前都从主内存刷新。因此，可以使用 `volatile` 来保证多线程操作时变量的可见性。除了 `volatile`，Java 中的 `synchronized` 和 `final` 两个关键字也可以实现可见性。

### 有序性

在 Java 中，可以使用 `synchronized` 和 `volatile` 来保证多线程之间操作的有序性。实现方式有所区别：

`volatile` 关键字会禁止指令重排。`synchronized` 关键字保证同一时刻只允许一条线程操作。

### 1.5.16 微服务组件 (Microservice Component)

Spring Cloud 是实施微服务的一系列套件，包括：服务注册与发现 (Spring Cloud Eureka<sup>52</sup>)、断路器 (Spring Cloud Hystrix)、服务状态监控、配置管理 (Spring Cloud Config)、智能路由、一次性令牌、全局锁、分布式会话管理、集群状态管理等。

目前的服务注册与发现组件主要有 Eureka、Zookeeper、Consul，Spring Cloud Eureka 是对 Netflix 的 Eureka 的进一步封装。Netflix 公司提供了包括 Eureka、Hystrix、Zuul、Archaius 等在内的很多组件，在微服务架构中至关重要，Spring 在 Netflix 的基础上，封装了一系列的组件，命名为：Spring Cloud Eureka、Spring Cloud Hystrix、Spring Cloud Zuul 等。客户端发现模式的方式是服务消费者选择合适的节点进行访问服务生产者提供的数据，这种选择合适节点的过程就是 Spring Cloud Ribbon 完成的。具体的使用 Ribbon 调用服务的话，你就可以感受到使用 Ribbon 的方式还是有一些复杂，因此 Spring Cloud Feign 应运而生。

Spring Cloud Feign 是一个声明 web 服务客户端，这使得编写 Web 服务客户端更容易，使用 Feign 创建一个接口并对它进行注解，它具有可插拔的注解支持包括 Feign 注解与 JAX-RS (Java API for RESTful Web Services<sup>53</sup>) 注解，Feign 还支持可插拔的编码器与解码器，Spring Cloud 增加了对 Spring MVC 的注解，Spring Web 默认使用了 `HttpMessageConverters`，Spring Cloud 集成 Ribbon 和 Eureka 提供的负载均衡的 HTTP 客户端 Feign。如果不用 Feign，微服务之间的调用可能像这样：

```
restTemplate.getForEntity("http://COMPUTE-SERVICE/add?a=10&b=20",  
    String.class).getBody()
```

使用了 Feign 之后，微服务之间的调用在写法上，表面上看起来更像是本地方法调用，减少了编码成本，代码可读性更高。Feign 的简洁明了方法调用写法：

```
restController.getForEntity(a,b)
```

简单的可以理解为：Spring Cloud Feign 的出现使得 Eureka 和 Ribbon 的使用更为简单。Spring Cloud Hystrix 正是为了解决这种情况的，防止对某一故障服务持续进行访问。Hystrix 的含义是：断路器，断路器本身是一种开关装置，用于我们家庭的电路

<sup>52</sup>Eureka 据说有一个含义是：我想到了。相传当年阿基米德看到浴缸溢出来的水，大喊“尤里卡”(Eureka)，意思是“我想到了”。这个说法来源之一是霍金自传，不知道 Netflix 团队在为项目起名字时是不是这个含义呢？。

<sup>53</sup>来源：[https://en.wikipedia.org/wiki/Java\\_API\\_for\\_RESTful\\_Web\\_Services](https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services)

Table 1.7: Spring Cloud 组件

组件名称	
Eureka	Eureka 用于服务的注册与发现。Eureka 是 Netflix 开源的一款提供服务注册和发现的产品，它提供了完整的 Service Registry 和 Service Discovery 实现。也是 spring Cloud 体系中最重要最核心的组件之一。
Ribbon	Ribbon 是 Netflix 发布的云中间层服务开源项目，其主要功能是提供客户端实现负载均衡算法。
Feign	Feign 支持服务的调用以及均衡负载。Feign 是从 Netflix 中分离出来的轻量级项目，能够在类接口上添加注释，成为一个 REST API 客户端。
Config	Spring Cloud Config 为分布式系统中的外部配置提供服务器和客户端支持。
Bus	Spring Cloud Bus 通过轻量消息代理连接各个分布的节点。
Stream	Spring Cloud Stream 是一个构建消息驱动微服务的框架。
Zuul	微服务网关。
Hystrix	Hystrix 处理服务的熔断防止故障扩散。Hystrix-dashboard 是一款针对 Hystrix 进行实时监控的工具，通过 Hystrix Dashboard 我们可以在直观地看到各 Hystrix Command 的请求响应时间, 请求成功率等数据。
Sleuth	Spring Cloud Sleuth 为服务之间调用提供链路追踪。通过 Sleuth 可以很清楚的了解到一个服务请求经过了哪些服务，每个服务处理花费了多长。从而让我们可以很方便的理清各微服务间的调用关系。

保护，防止电流的过载，当线路中有电器发生短路的时候，断路器能够及时切换故障的电器，防止发生过载、发热甚至起火等严重后果。

需要将权限控制这样的东西从我们的服务单元中抽离出去，而最适合这些逻辑的地方就是处于对外访问最前端的地方，我们需要一个更强大一些的均衡负载器，它就是本文将来介绍的：服务网关。服务网关是微服务架构中一个不可或缺的部分。通过服务网关统一向外系统提供 REST API 的过程中，除了具备服务路由、均衡负载功能之外，它还具备了权限控制等功能。Spring Cloud Netflix 中的 Zuul 就担任了这样一个角色，为微服务架构提供了前门保护的作用，同时将权限控制这些较重的非业务逻辑内容迁移到服务路由层面，使得服务集群主体能够具备更高的可复用性和可测试性。

### 1.5.17 ThreadLocal 共享认证 Token

Token 需要在拦截器里面添加到请求头中，在启动定时任务时设置好 Token，在 HTTP 请求和 Websocket 连接时，取出传入服务端：

```
1 public class RobotAuthTokenHandler {  
2     private static ThreadLocal<String> authToken = ThreadLocal.<  
3         String>withInitial(() -> {  
4             return null;  
5         });  
6  
7     public String getAuthToken() {  
8         return authToken.get();  
9     }  
10  
11     public void setAuthToken(String value) {  
12         authToken.set(value);  
13     }  
14 }
```

使用 ThreadLocal 在线程内共享变量实现优雅，否则需要在方法间传递参数，如何将参数一步一步传递到拦截器里？或者使用全局静态变量来共享，但是需要处理线程间共享变量问题，又涉及到锁，线程安全等等问题。



### 1.5.18 科学上网之 Amazon Could 篇 (Fenceless Internet)

没有 Google，检索资料很是不便，1 个搬瓦工主机，2 个 Google Cloud 主机通阵亡 (2019 年 6 月 20 日)。2018 年底墙体做了加固升级改造，其中现在 IP 地址封锁 (不允许到那家串门)，IP 端口封锁 (不允许从某一个门进)，TCP 关键字阻断 (不许谈论某些话题，具体什么话题根据具体情况来确定，一般情况下政治和宗教问题要避免谈论，一般查询技术相关信息不需要特别留意)，CA 证书检测技术已经领跑全球。现在的围墙应该会采集数据特征，通过一系列算法进行特征分析，做到更智能，以前可能需要人工干预，现在会根据流量特征自动化识别和阻断，也算是大数据和 AI 应用的一个范例。举个例子，IP 阻断，换做以前怎么操作，手工添加？全球这么多 IP，是封锁不过来的。现在只要触发规则，有多少封多少，可以极大的提高效率解放人力，其中的过程完全是自动化的。现在运营商即使不知道所有的内容，但是通信的绝大部分信息是知道的，所以扶墙行为一定要避开敏感内容 (政治 + 宗教)。

还好搬瓦工的密码还记得，本来想将搬瓦工的 VPS (Virtual Private Server) 迁移到其他到数据中心 (Migrate to another datacenter) 暂时解决燃眉之急，迁移后 IP 地址会相应的变化，也就解决了 IP 被封的问题。不过迁移的时候提示错误：Migration backend is currently not available for this VPS. Please try again in 10-15 minutes. (734152)。这个是由于搬瓦工 IP 被 Block 了之后，不能迁移<sup>54</sup>。又想购买新的 VPS，结果发现好多购买了 VPS 不久后就立即被封了，此种方法不够保险。最终采用 Amazon 的免费套餐自己搭建隧道来实现<sup>55</sup>，讲真搭建这个梯子真的花掉太多时间，除了练习不同的姿势，大部分时间都被浪费掉了。多数时候真的是得不偿失，但是当没有足够丰富的信息源时，就没有办法高效的工作，效率那是要降低不少，弯路要走很多，不断的分辨无效的信息 (相信百度能够做到在保障用户体验下也能够捞金)，毕竟作为资质平庸的程序员不可能记住所有的东西。肯定要毫不客气的踩在别人的肩膀上。启动客户端时，使用 verbose 模式可以输出本地 Shadowsocks 代理的详细日志信息，可以检查到底代理配置是否已经生效：

```
1 # v 参数表示 verbose 模式
2 # 可查看详细日志输出
3 ss-local -v -c /etc/shadowsocks/shadowsocks.json
```

Amazon Cloud 服务端配置如下，在被封锁的情况下，可以尝试重新启动实例或者

<sup>54</sup>来源链接：<https://www.bandwagonhost.net/3174.html>

<sup>55</sup>参考教程：<https://www.eeebe.com/article/ce37aebf65afbef8.html>

删除原有实例，安装新的实例 (注意重启实例之后，链接的域名也可能发生变化，重新启动更换 IP 后需要调整本地的域名)：

```
1 {  
2     "server": "0.0.0.0",  
3     "server_port": 443,  
4     "local_address": "127.0.0.1",  
5     "local_port": 1080,  
6     "password": "password",  
7     "timeout": 300,  
8     "method": "aes-256-cfb",  
9     "fast_open": false,  
10    "workers": 1  
11 }
```

在 Ubuntu 主机上，服务端文件/etc/systemd/system/shadowsocks.service 中添加如下配置，将 Shadowsocks Server 端做成自动启动端方式：

```
1 [Unit]  
2 Description=Shadowsocks Server Service  
3 After=network.target  
4  
5 [Service]  
6 Type=simple  
7 User=root  
8 ExecStart=/usr/local/bin/ssserver -c /etc/shadowsocks/shadowsocks.  
    json  
9  
10 [Install]  
11 WantedBy=multi-user.target
```

开始时，指定服务端的默认端口为 443 端口，提示错误：unsupported addrtype 190, maybe wrong password or encryption method, can not parse header when handling connection from 209.117.196.186:57090。重新调整服务端的端口为 2233 即可解决。有

时 SwitchOmega 代理不生效，只有系统代理生效，重新安装 SwitchOmega 解决。由于 Google Chrome 逐渐封闭策略，无法直接安装时，解压 crx 文件后，通过开发者模式加载解压后文件夹来安装。

### 1.5.19 IntelliJ Idea 常用技巧 (IntelliJ Idea tricks and tips)

#### 自动优化导入包

Settings->Editor->General->Auto Import；选中 Optimize imports on the fly 和 Add unambiguous imports on the fly。Optimize imports on the fly：自动去掉一些没有用到的包；Add unambiguous imports on the fly：自动帮我们优化导入的包。

#### 创建对象

例如需要创建一个新的 User 对象，不需要直接输入 `User user=new User()`，可以直接输入 `new User().var`，再按下 Tab 按钮即可。非空判断时，可以直接输入 `user.null`，按下 Tab 按钮，即可自动生成非空判断语句，`notnull` 同理。

#### 智能提示大小写不敏感

CSharp 的 string 是小写，Java 的 string 是大写，在 IntelliJ Idea 中输入小写的字母，默认不提示大写开头的类，有那么一点点不方便。键入代码忽略大小写，在 2019.01 版本之前的 IDE 设置 code completion 的 case sensitive completion，你只需要把后面选择为 none 就可以。2019.01 开始的版本在 code completion 设置 match case。

#### 显示 Run Dashboard

在微服务开发时，相应的微服务目前有将近 10 个，每次启动时特别不方便，此时可以用 Dashboard 来显示所有微服务的状态 (注意从 IntelliJ Idea Ultimate 2019.02 版本开始，Run Dashboard 默认显示名称为 Services)。在 IntelliJ Idea 中启用 Run Dashboard，在当前项目的 idea 隐藏文件夹下，找到 workspace.xml<sup>56</sup>文件，RunDashboard 节下添加如下配置：

```
1 <option name="configurationTypes">
2   <set>
3     <option value="SpringBootApplicationConfigurationType" />
4   </set>
```

<sup>56</sup>此文件为用户 IDE 环境配置文件，该文件存储个人设置，例如窗口位置，以及其它附属于开发环境的信息，一般情况下不建议存放到 VCS 中

```
</option>
```

配置添加后,就可以在 IntelliJ Idea 的 View->Tool Window 下看到 Run Dashboard 菜单了,如图所示。一般情况下,微服务可能有几个甚至几十,几百个,在 Run Dashboard 的一级目录上点击启动,即可一键启动所有微服务。遗憾的是,现在还没有找到定义服务启动顺序的方法,无法定义服务的启动顺序。

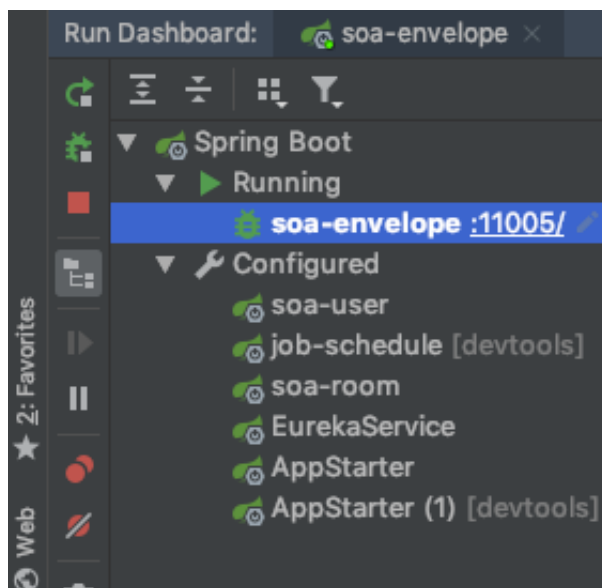


Figure 1.12: JRun Dashboard 面板

### 1.5.20 Mac 快捷键

### 1.5.21 youtube 下载

安装 youtube-dl:

```
sudo curl -L https://yt-dl.org/downloads/latest/youtube-dl -o /usr  
/local/bin/youtube-dl
```

添加可执行权限:

Table 1.8: 快捷键

IntelliJ Idea 快捷键	作用	备注
Ctrl + Option + O(Optimize)	优化导入	
Option + Command + L)	格式化代码	
Command + Shift + .	显示隐藏文件	
Command + -	折叠代码	
Command + +	展开代码	

Table 1.9: 快捷键

Mac 快捷键	作用	备注
Option + Command + Esc	打开强制退出应用窗口	

```
1 sudo chmod a+rx /usr/local/bin/youtube-dl
```

查看可下载视频格式：

```
1 youtube-dl -F https://www.youtube.com/watch?v=nElPx2RrxXw
```

下载视频：

1.6 06 月

1.6.1 RabbitMQ Docker 部署

RabbitMQ 还有 ARM 版本，意味者可以安装在树莓派这种设备上。但是一般安装还是选择支持 X86 指令集的版本，选择带有“mangement”的版本（包含 web 管理页面），拉取镜像，当前安装的时间是 2019 年 06 月 19 日，最新的版本是 3.7.15，注意不同的时间安装不同版本的镜像：

```
1 # 拉取镜像
2 docker pull rabbitmq:3.7.15-management
```

```
3 # 查看镜像
4 docker images
```

根据下载的镜像创建和启动容器：

```
1 # 运行容器
2 # 用斜线隔开命令时，后一个命令与斜线中间不要有空格
3 docker run -d --name rabbitmq3.7.7 -p 5672:5672 -p 15672:15672 \
4 -v 'pwd' /data:/var/lib/rabbitmq --hostname myRabbit \
5 -e RABBITMQ_DEFAULT_VHOST=my_vhost \
6 -e RABBITMQ_DEFAULT_USER=admin \
7 -e RABBITMQ_DEFAULT_PASS=fJ8belfM1H6P7ymF f05c3eb3cf91
```

-d(deamon) 指定后台运行容器，-name 指定容器名，一般情况下推荐指定容器名称，往后操作容器可以根据易于记忆的名称来作为标识，省去了查看容器 ID 的步骤，-p(port) 指定服务运行的端口（5672：应用访问端口；15672：控制台 Web 端口号），-v(volume) 映射目录或文件；-hostname 主机名（RabbitMQ 的一个重要注意事项是它根据所谓的“节点名称”存储数据，默认为主机名），-e(environment) 指定环境变量；（RABBITMQ\_DEFAULT\_VHOST：默认虚拟机 (Virtual Host) 名；RABBITMQ\_DEFAULT\_USER：消息队列默认的用户名；RABBITMQ\_DEFAULT\_PASS：消息队列默认密码）。6ffc11daa8d0 是 RabbitMQ 镜像的 ID。

### 1.6.2 Websocket 性能测试

Websocket 性能测试时主要关注 2 个指标，一个是单台服务器可以同时建立多少 Websoket 连接（简单测试后，阿里云 4 核 16GB 可以建立 6000+ 连接，由于 6000+ 已经满足目前的需求，没有再继续新增连接测试最大连接数），二是建立连接后，消息发送的延迟情况。单台服务器并发连接数采用了一个简单的 HTML 页面，同时打开指定数量的连接，需要注意的是浏览器对 Websocket 的连接数量有限制（2019 年 06 月 18 日时 Google Chrome 的限制在 250 个左右，所采用的 Google Chrome 版本是 Version 74.0.3729.169 (Official Build) (64-bit)，测试客户端操作系统是 Mac OS Mojave），测试的时候可以开多个浏览器来增加连接数量。

```
1 <!DOCTYPE HTML>
2 <html>
```

```
3
4 <head>
5   <meta charset="utf-8">
6   <title>runoob.com</title>
7   <script type="text/javascript">
8     function WebSocketTest() {
9       if ("WebSocket" in window) {
10         for (let i = 0; i < 250; i++) {
11           newcon();
12         }
13       }
14     }
15
16     function newcon() {
17       var ws = new WebSocket("ws://localhost:8070");
18       console.log("open" + i + "websocket");
19       ws.onopen = function () {
20         ws.send("发送数据");
21       };
22
23       ws.onmessage = function (evt) {
24         var received_msg = evt.data;
25         console.log(received_msg);
26
27       };
28
29       ws.onclose = function () {
30         // 关闭 websocket
31         console.log("closed");
32       };
33     }
34   </script>
35 </head>
```

```
36
37 <body>
38   <div id="sse">
39     <a href="javascript:WebSocketTest()">Run WebSocket</a>
40   </div>
41 </body>
42
43 </html>
```

使用脚本的好处是，一旦建立了 Websocket 连接之后，在不刷新页面或者不关闭浏览器页面的情况下，会话可以一直保持。而用 JMeter 测试 Websocket 时，由于 JMeter 是通过开启线程来连接 Websocket，当线程销毁后，JMeter 与服务端的会话后立即就关闭了。另一种更加合理的方式是使用 Java(也可以换成 Python 等其他任意一种语言，只要对应的 Websocket 库支持到位) 写一个简单的客户端<sup>57</sup>，单个客户端的并发连接数量可以支持到 1300 左右 (实际是 1355 个)。

### 调整 Nginx 的 worker\_connections

开启了 500 个以上的 Websocket 连接时，Nginx 提示错误 worker\_connections are not enough while connecting to upstream。设置工作进程可以打开的最大并发连接数。:

```
1 句法: worker_connections number;
2 默认: worker_connections 512;
3 语境: events
```

这个数字包括所有连接（例如与代理服务器的连接等，有反向代理链接数就会有相应增加），而不仅仅是与客户端的连接。另一个考虑因素是，同时连接的实际数量不能超过打开文件的最大数量的当前限制。

### 1.6.3 用 curl 测试接口

curl 以命令行交互的方式传输数据，支持 HTTP, HTTPS, FTP, FTPS, SCP, SFTP, TFTP, DICT, TELNET, LDAP or FILE 协议。使用 curl 非常方便，几乎所有的操作系统 () 都包含此工具，Windows 例外，不过最新的微软改变了以往封闭的做法，Windows

<sup>57</sup>相关问答参考: [https://stackoverflow.com/questions/56652173/is-it-possible-to-create-multi-websocket-client-in-one-html?noredirect=1#comment99913974\\_56652173](https://stackoverflow.com/questions/56652173/is-it-possible-to-create-multi-websocket-client-in-one-html?noredirect=1#comment99913974_56652173)



已经逐步集成了 Linux Kernel。所以从萨提亚·纳德拉 (Satya Nadella) 执掌微软后, Windows 也会以特定的方式集成 curl。使用 curl 命令请求接口, 带上认证的消息头 (Header):

## GET

测试请求接口:

```
1 curl http://localhost:11004/room/1 -H "token:1" | jq '.'
```

```
2 curl http://localhost:11003/robot/1 -H "token:1" | jq '.'
```

```
3 curl http://localhost:11002/wallet/consume/1 -H "token:1" | jq '.'
```

```
4 # 进入房间
```

```
5 curl -X PUT --header 'Content-Type: application/json' \
```

```
6 --header 'Accept: application/json' --header 'appCode: 101' \
```

```
7 --header 'appVersion: Unknown' \
```

```
8 --header 'clientId: 7c54ff40-f11e-41c0-8d31-4b26d6569b5a' \
```

```
9 --header 'requestId: 2e0dddf9-17d9-42d9-8614-b44de17fe135' \
```

```
10 --header 'token: 1609c5bf-dd2c-4667-a7da-32926cb6847d' \
```

```
11 'https://localhost:11004/envelope/room/match/1/1'
```

## POST

测试保存接口:

```
1 curl -X POST http://localhost:11002/wallet -H "content-
```

```
    type:application/json;charset=utf-8" \
```

```
2 -H "token:1" \
```

```
3 -d '{"id":2,"userId":1,"remainNum":1,"createTime":1234567,"
```

```
    updateTime":1234567,"activeNum":4,"frozenNum":1,"
```

```
    maxLimitPerday":5,"totalLimit":8,"moneyType":1,"appId":
```

```
    "0010010002"}' | jq '.'
```

```
4
```

```
5 curl -X POST http://localhost:11002/wallet/consume -H "content-
```

```
    type:application/json;charset=utf-8" \
```

```
6 -H "token:1" \
```

```
7 -d '{"id":2,"roomId":1,"consumeNum":1,"consumeItem":"红包游戏","
    userId":1,"consumeTime":123,"description":"demo","appId":
    "0010010001","transNo":"1","remainNum":234,"currentNum"
    :232,"currentStatus":1,"payType":1,"createTime":232432,"
    updateTime":232343}' | jq '.'
```

## PUT

测试修改 (PUT) 类型接口, 请求头中指定 Content-Type 为 application/json, 以 Json 的方式将实体对象提交给后端, 请求里面带上用户的认证信息:

```
1 curl -X PUT http://localhost:11005/account/charge \
2 -H "Content-Type: application/json" \
3 -H "token:user:login:token:56ac2f34-bf19-442d-959a-ea31784dcd7c" \
4 -d '{"userId":1,"amount":1000}'
```

### 1.6.4 扫描如何排除特定类

Spring 启动项目时, 提示错误:

```
1 Failed to configure a DataSource: 'url' attribute is not specified
    and no embedded datasource could be configured.
2 Reason: Failed to determine a suitable driver class
```

是由于依赖中包含数据库组件, 启动的时候注解指定排除即可。

```
1 @SpringBootApplication(exclude = { DruidDataSourceAutoConfigure.
    class })
```

此处指定的数据库连接组件是 Druid, 排除类 DruidDataSourceAutoConfigure。需要排除其他组件直接在 exclude 中指定, 以逗号隔开即可。

### 1.6.5 网站启用 https

Let's Encrypt 是一个于 2015 年三季度推出的数字证书认证机构, 旨在以自动化流程代替手动证书管理, 并推广万维网网站使用加密连接, 为网站提供免费的 SSL/TLS

证书<sup>58</sup>。Certbot 是专门为 Let's Encrypt 制作的一个管理证书工具，可以通过它来生成证书以及管理更新 Let's Encrypt 证书。自动管理证书根据 ACME 协议来进行，那为什么要创建 ACME(Automated Certificate Management Environment) 协议呢，传统的 CA(Certificate Authority) 机构是人工受理证书申请、证书更新、证书撤销，完全是手动处理的。而 ACME 协议规范化了证书申请、更新、撤销等流程，只要客户端实现该协议的功能，通过客户端就可以向 Let's Encrypt 申请证书，也就是说 Let's Encrypt CA 完全是自动化操作的。任何人都可以基于 ACME 协议实现一个客户端，官方推荐的客户端是 Certbot。安装证书管理客户端 Certbot<sup>59</sup>(版本：0.35.1):

```
1 wget https://dl.eff.org/certbot-auto
2 sudo mv certbot-auto /usr/local/bin/certbot-auto
3 sudo chown root /usr/local/bin/certbot-auto
4 chmod 0755 /usr/local/bin/certbot-auto
```

客户端安装完毕后，运行如下命令获取证书：

```
1 /usr/local/bin/certbot-auto certonly --standalone -d rancher.
   example.com
```

按照提示操作，如果最终提示 congratulations，则表示证书生成成功。证书的存放位置一般在/etc/letsencrypt/archive 目录下。但是配置完毕后，在客户端无法访问 URL，Nginx 日志中提示 upstream sent no valid HTTP/1.0 header while reading response header from upstream 错误。使用如下命令在内网访问服务：

```
1 curl -k https://172.19.4.31:4431|jq '.'
```

curl 使用 k 参数 (-insecure) 表示允许没有证书的情况下连接 SSL 站点 (Allow connections to SSL sites without certs)。通过外部域名查看服务响应：

```
1 curl -k https://rancher.example.com|jq '.'
```

---

<sup>58</sup> 参考链接：[https://zh.wikipedia.org/wiki/Let%27s\\_Encrypt](https://zh.wikipedia.org/wiki/Let%27s_Encrypt)

<sup>59</sup> 官方文档：<https://certbot.eff.org/docs/install.html>

结果是内网可以正常访问服务，外部网络无法访问，问题定位在 Nginx 内部转发的过程，发现是 proxy\_pass 转发到内网节点时时默认使用的 HTTP 协议 (内网服务默认是 HTTPS)，调整为 HTTPS 即可。

### 通配符证书 (Wildcard Certificate)

在计算机网络中，通配符证书是一个可以被多个子域 (Subdomain) 使用的公钥证书。通配符证书的优点是节省了购买证书的成本，提高了配置证书的便捷性。任何客户端只要支持 ACME v2 版本，就可以申请通配符证书，官方介绍 Certbot 0.22.0 版本支持新的协议版本 (也就是支持通配符证书的 ACME v2 版本)，在 2018 年 03 月 14 日正式宣布支持通配符证书<sup>60</sup>(Wildcard Certificate)。需要注意的是，目前 (2019 年 06 月 11 日) 通配符证书仅仅支持一级子域，例如域名 pay.example.com 支持，但是域名 seller.pay.example.com 不支持。应用通配符证书步骤如下：

#### Step 1: 申请通配符证书：

```
1 /usr/local/bin/certbot-auto certonly -d *.example.com --manual --  
    preferred-challenges dns --server https://acme-v02.api.  
    letsencrypt.org/directory
```

**Step 2:** 在阿里云域名配置界面，配置 DNS TXT 记录，从而校验域名所有权，也就是判断证书申请者是否有域名的所有权。如下命令验证 DNS TXT 记录是否生效：

```
1 dig -t txt _acme-challenge.example.com @8.8.8.8
```

如果 ANSWER SECTION 中包含添加的 TXT 记录，那么表示添加的域名映射生效，确定生效后，按 Enter 继续执行 Step 1 证书申请命令。输出内容出现 congratulations 提示表示证书签发成功。然后利用 openssl 工具<sup>61</sup>手工查看证书信息，输入如下命令：

```
1 openssl x509 -in /etc/letsencrypt/archive/example.com/cert1.pem -  
    noout -text
```

<sup>60</sup>信息来源: <https://www.infoq.cn/article/2018/03/lets-encrypt-wildcard-https>

<sup>61</sup>它是 SSL/TLS 的一个实现，而且包括了一批非常棒的工具软件。

从输出的结果可以看出，证书包含了 SAN(Subject Alternative Name) 扩展，该扩展的值就是 \*.example.com。

**Step 3:** 配置 Nginx 使用 SSL 证书，在 Nginx 中指定 PEM(Privacy Enhanced Mail) 证书路径：

```
1 server{
2     listen 443 ssl;
3     ssl_certificate /etc/nginx/conf.d/cert/example.com/fullchain1.
        pem;
4     ssl_certificate_key /etc/nginx/conf.d/cert/example.com/
        privkey1.pem;
5     server_name apollo.example.com;
6 }
```

一般 Apache 和 Nginx 服务器应用偏向于使用 PEM 这种编码格式。Java 和 Windows 服务器应用偏向于使用 DER 这种编码格式<sup>62</sup>。网站统一 HTTPS 访问，将 80 端口的请求 HTTP 重定向到 HTTPS：

```
1 rewrite ^(.*)$ https://${server_name}$1 permanent;
```

### 证书续期 (Certificate Renew)

截止 2019 年 09 月 1 日，Let's Encrypt 证书默认是 90 天过期，输入如下命令对证书进行续期：

```
1 ./certbot-auto --server https://acme-v02.api.letsencrypt.org/
    directory \
2 -d "*.ttt208.com" -d "ttt208.com" \
3 --manual --preferred-challenges dns-01 certonly
```

certbot-auto 执行证书续期后，注意默认对存放路径在 etc/letsencrypt/live，如果和 nginx 读取证书的路径不一致，需要拷贝替换掉旧有的证书文件。查看证书过期可以直接在浏览器查看，也可以通过如下的命令查看证书的有效时间范围：

<sup>62</sup>来源：<https://kangzubin.com/certificate-format/>

```
1 openssl x509 -in cert1.pem -noout -dates
```

### 1.6.6 MySQL 事务隔离级别与锁

Read Uncommitted (读取未提交内容) 在该隔离级别, 所有事务都可以看到其他未提交事务的执行结果<sup>63</sup>。本隔离级别很少用于实际应用, 因为它的性能也不比其他级别好多少。读取未提交的数据, 也被称之为脏读 (Dirty Read)。Read Committed (读取提交内容) 这是大多数数据库系统的默认隔离级别 (但不是 MySQL 默认的)。它满足了隔离的简单定义: 一个事务只能看见已经提交事务所做的改变。这种隔离级别也支持所谓的不可重复读 (Nonrepeatable Read), 因为同一事务的其他实例在该实例处理其间可能会有新的 commit, 所以同一 select 可能返回不同结果。Repeatable Read (可重复读) 这是 MySQL 的默认事务隔离级别, 它确保同一事务的多个实例在并发读取数据时, 会看到同样的数据行。不过理论上, 这会导致另一个棘手的问题: 幻读 (Phantom Read)。简单的说, 幻读指当用户读取某一范围的数据行时, 另一个事务又在该范围内插入了新行, 当用户再读取该范围的数据行时, 会发现有新的“幻影”行。InnoDB 和 Falcon 存储引擎通过多版本并发控制 (MVCC, Multiversion Concurrency Control) 机制解决了该问题。Serializable (可串行化) 这是最高的隔离级别, 它通过强制事务排序, 使之不可能相互冲突, 从而解决幻读问题。简言之, 它是在每个读的数据行上加上共享锁。在这个级别, 可能导致大量的超时现象和锁竞争。

### 1.6.7 垃圾回收 (Garbage Collection)

内存管理是 C/C++ 相当令人头疼的问题, C/C++ 高手从中获得了更大的自由度和更好的性能, 而新手很容易陷入一遍又一遍的抓狂<sup>64</sup>。想成为 C/C++ 高手, 内存管理是一门基本功课。C 语言手动释放内存如下代码片段所示<sup>65</sup>:

```
1 int send_request() {  
2     size_t n = read_size();  
3     int *elements = malloc(n * sizeof(int));  
4     if(read_elements(n, elements) < n) {  
5         // elements not freed!
```

<sup>63</sup>[http://xstarcd.github.io/wiki/MySQL/mysql\\_isolation\\_level.html](http://xstarcd.github.io/wiki/MySQL/mysql_isolation_level.html)

<sup>64</sup>参考来源: <https://chenqx.github.io/2014/09/25/Cpp-Memory-Management/>

<sup>65</sup>示例来源: <https://www.ktanx.com/blog/p/2952>

```
6         return -1;
7     }
8     free(elements)
9     return 0;
10 }
```

但是现实情况是，很多时候没有注意或者忘记释放，无意间就造成程序内存泄漏 (Memory Leak)。这种感觉就像，出门之后突然想起来没有锁门。只能一遍一遍的提醒自己，记得锁门、记得锁门，对于部分选手来说，遇到如此糟糕的境况很难完全避免。为了解决手动管理的问题，John McCarthy 提出了标记-清除 (Mark Sweep) 垃圾回收算法，使得编程人员不用操心内存回收。John McCarthy 身为 Lisp<sup>66</sup>之父和人工智能之父，同时，他也是 GC 之父。1960 年，他在其论文<sup>67</sup>中首次发布了 GC 算法 (作为对比，世界上第一台通用计算机“ENIAC”于 1946 年 2 月 14 日诞生，Lisp 语言首次出现是在 1958 年，Unix 操作系统首次发布的时间是 1971 年 11 月 3 日，C 语言首次出现是在 1972 年)。而 Java 的前身 Oak 是在 1990 发布的，利用 JVM 实现了跨平台。垃圾回收这个概念在 20 世纪 60 年代出现，所提出来都处理方式过去了半个多世纪了还在沿用。目前有 GC 机制的语言有 Lisp、Java、Ruby、Python、Perl、Haskell。

- 1959: D. Edwards 实现了 GC
- 1960: John McCarthy 发布了初代 GC 算法即标记-清除 (Mark Sweep) 算法
- 1960: George E. Collins 发布了引用计数算法
- 1963: Marvin L. Minsky 发布了复制算法
- 1996: 首次出版了 Garbage Collection 一书

### 1.6.8 JVM 运行时数据区 (JVM Run-time Data Areas)

Java 虚拟机运行时的数据区定义也是 Java 内存模型 (Java Memory Model) 的定义。JVM 加载执行 Java 程序的过程中会把它所管理的内存划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而存在，有些区域则是依赖用户线程的启动和结束而建立和销毁<sup>68</sup>。根据 JVM 虚拟机

<sup>66</sup>Lisp (历史上拼写为 LISP) 是具有悠久历史的计算机编程语言家族，有独特和完全括号的前缀符号表示法。起源于公元 1958 年，是现今第二悠久而仍广泛使用的高端编程语言。参考链接：<https://zh.wikipedia.org/wiki/LISP>

<sup>67</sup><http://www-formal.stanford.edu/jmc/recursive/node4.html#tex2html8>

<sup>68</sup>参考 Java SE 8 虚拟机规范<https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.5>

规范，运行时数据区通常包括这几个部分：程序计数器 (Program Counter Register)、Java 栈 (VM Stack)、本地方法栈 (Native Method Stack)、方法区 (Method Area)、堆 (Heap)。如图1.13所示：

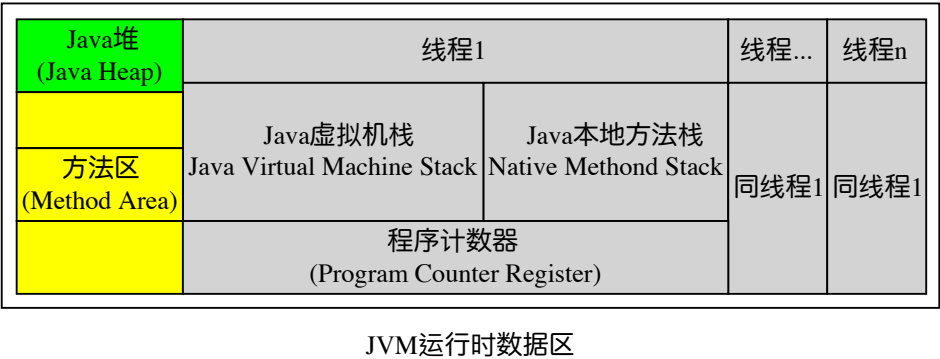


Figure 1.13: JVM 运行时数据区

**程序计数器 (Program Counter Register)** 是一个记录着当前线程所执行的字节码的行号指示器。JVM 的多线程是通过 CPU 时间片轮转（即线程轮流切换并分配处理器执行时间）算法来实现的。也就是说，某个线程在执行过程中可能会因为时间片耗尽而被挂起，而另一个线程获取到时间片开始执行。当被挂起的线程重新获取到时间片的时候，它需要从被挂起的地方继续执行，就必须知道它上次执行到哪个位置，在 JVM 中，通过程序计数器来记录某个线程的字节码执行位置。因此，程序计数器是具备线程隔离的特性，也就是说，每个线程工作时都有属于自己的独立计数器。如果当前线程正在执行 Java 方法，则程序计数器保存的是虚拟机字节码的内存地址，如果正在执行的是 Native 方法（非 Java 方法，JVM 底层有许多非 Java 编写的函数实现），计数器则为空。程序计数器是唯一一个在 Java 规范中没有规定任何 OutOfMemory 场景的区域。

**方法区 (Method Area)** 在 JVM 中也是一个非常重要的区域，它与堆一样，是被线程共享的区域。在方法区中，存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等。

**虚拟机栈 (VM Stack)**：每个线程有一个私有的栈，随着线程的创建而创建。栈里面存着的是一种叫“栈帧”的东西，每个方法会创建一个栈帧，栈帧中存放了局部变量表（基本数据类型和对象引用）、操作数栈、方法出口等信息。栈的大小可以固定也可



以动态扩展。当栈调用深度大于 JVM 所允许的范围，会抛出 `StackOverflowError` 的错误。

注意，虚拟机规范并不是一成不变的，Oracle 在发布新的 JAVA 版本时，可能会对 JVM 做一定的优化和改进，例如在 JDK 8 的版本中，方法区被移除，取而代之的是 `Metaspace`（元数据空间）。JVM 中的堆（Heap），一般分为三大部分：新生代、老年代、永久代：新生代又分为 `Eden` 区、`SurvivorFrom`、`SurvivorTo` 三个区。`Eden` 区：Java 新对象的出生地（如果新创建的对象占用内存很大，则直接分配到老年代）。当 `Eden` 区内内存不够的时候就会触发 `MinorGC`，对新生代区进行一次垃圾回收。`SurvivorTo`：保留了一次 `MinorGC` 过程中的幸存者。`SurvivorFrom`：上一次 GC 的幸存者，作为这一次 GC 的被扫描者。

### 1.6.9 JVM 方法区

方法区与堆一样，是被线程共享的区域。在方法区中，存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等。在 `Class` 文件中除了类的字段、方法、接口等描述信息外，还有一项信息是常量池<sup>69</sup>（`Class Constant Pool`），用来存储编译期间生成的字面量<sup>70</sup>（`Literal`）和符号引用<sup>71</sup>（`Symbolic References`）。在方法区中有一个非常重要的部分就是运行时常量池（`Runtime Constant Pool`），它是每一个类或接口的常量池的运行时表示形式，在类和接口被加载到 JVM 后，对应的运行时常量池就被创建出来。当然并非 `Class` 文件常量池中的内容才能进入运行时常量池，在运行期间也可将新的常量放入运行时常量池中，比如 `String` 的 `intern` 方法。

### 1.6.10 Docker 容器与宿主机通信

Docker 容器的默认网络模式是桥接模式（`bridge`），桥接模式为容器创建独立的网络命名空间，容器具有独立的网卡等所有单独的网络栈。通过宿主机上的 `docker0` 网

---

<sup>69</sup><https://tangxman.github.io/2015/07/27/the-difference-of-java-string-pool/>

<sup>70</sup>字面量就是我们所说的常量概念，如文本字符串、被声明为 `final` 的常量值等。

<sup>71</sup>符号引用是一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可（它与直接引用区分一下，直接引用一般是指向方法区的本地指针，相对偏移量或是一个能间接定位到目标的句柄）。一般包括下面三类常量：

- 类和接口的全限定名
- 字段的名称和描述符
- 方法的名称和描述符

桥<sup>72</sup>，容器可以与宿主机以及外界进行网络通信，Docker 网桥在内核层连通了其他的物理或虚拟网卡 (Virtual Ethernet)，这就将所有容器和本地主机都放到同一个物理网络。Docker 默认指定了 docker0 接口的 IP 地址和子网掩码，让主机和容器之间可以通过网桥相互通信，它还给出了 MTU（接口允许接收的最大传输单元），通常是 1500 Bytes。网桥在安装完 Docker 后自动添加，添加网桥的同时，宿主机也会在内核路由表上添加一条到达相应网络的静态路由，可通过 `route -n` 命令查看，如图1.14所示：

```
[root@izbp19pke6x0v6ruecuy1yz ~]# route -n
```

Kernel IP routing table

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	172.17.63.253	0.0.0.0	UG	0	0	0	eth0
10.42.0.2	0.0.0.0	255.255.255.255	UH	0	0	0	cali9ebc5c6dee2
10.42.0.3	0.0.0.0	255.255.255.255	UH	0	0	0	cali89084a0aa09
10.42.0.4	0.0.0.0	255.255.255.255	UH	0	0	0	cali2f8ee076783
10.42.0.6	0.0.0.0	255.255.255.255	UH	0	0	0	cali513f783abf5
169.254.0.0	0.0.0.0	255.255.0.0	U	1002	0	0	eth0
172.17.0.0	0.0.0.0	255.255.192.0	U	0	0	0	eth0
172.18.0.0	0.0.0.0	255.255.0.0	U	0	0	0	docker0

Figure 1.14: Docker 主机默认路由

此条路由表示所有目的 IP 地址为 172.17.0.0/16 的数据包从 docker0 转发。由于目前 Docker 网桥是 Linux 网桥，用户可以使用 `brctl show` 命令来查看网桥和端口连接信息。在 CentOS 下，如果没有 `brctl` 命令，输入如下命令安装 `brctl`(Bridge Control):

```
1 yum install bridge-utils
```

容器内部报文发送到外部的流程为：

### 1.6.11 Docker 跨主机通信方案

Docker 默认的网络环境下，单台主机上的 Docker 容器可以通过 docker0 网桥直接通信，而不同主机上的 Docker 容器之间只能通过在主机上做端口映射进行通信。这种端口映射方式对很多集群应用来说极不方便。如果能让 Docker 容器之间直接使用自己的 IP 地址进行通信，会解决很多问题。按实现原理可分别直接路由方式、桥接方式（如 `pipework`）、Overlay 隧道方式（如 `flannel`、`ovs+gre`）等。Docker 跨主机通信组

<sup>72</sup>可以把它想象成一个虚拟的交换机，所有的容器都是连到这台交换机上面的。docker 会从私有网络中选择一段地址来管理容器，比如 172.17.0.1/16，这个地址根据你之前的网络情况而有所不同。

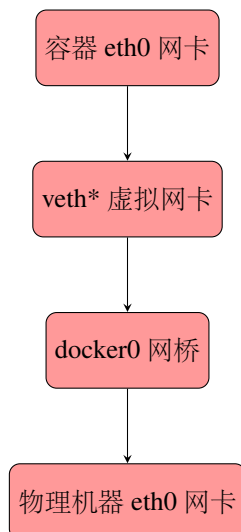


Figure 1.15: 容器与主机通信过程

件的比较<sup>73</sup>。

### overlay

Overlay 网络是指在不改变现有网络基础设施的前提下，通过某种约定通信协议，把二层报文封装在 IP 报文之上的新的数据格式。这样不但能够充分利用成熟的 IP 路由协议进程数据分发，而且在 Overlay 技术中采用扩展的隔离标识位数，能够突破 VLAN 的 4000 数量限制，支持高达 16M 的用户，并在必要时可将广播流量转化为组播流量，避免广播数据泛滥。因此，Overlay 网络实际上是目前最主流的容器跨节点数据传输和路由方案。Overlay 网络的实现方式可以有多种，其中 IETF（国际互联网工程任务组）制定了三种 Overlay 的实现标准<sup>74</sup>。

1. 虚拟可扩展 LAN（VXLAN）
2. 采用通用路由封装的网络虚拟化（NVGRE）
3. 无状态传输协议（SST）

Docker 内置的 Overlay 网络是采用 IETF 标准的 VXLAN 方式，并且是 VXLAN 中

<sup>73</sup><https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-april-2019-4a9886efe9c4>

<sup>74</sup>来源链接：<https://www.cnblogs.com/jicki/p/5548610.html>

普遍认为最适合大规模的云计算虚拟化环境的 SDN Controller 模式。Docker 的 Overlay 网络功能与其 Swarm 集群是紧密整合的，因此为了使用 Docker 的内置跨节点通信功能，最简单的方式就是采纳 Swarm 作为集群的解决方案。在 Docker 1.9 中，要使用 Swarm + overlay 网络架构，还需要以下几个条件：

1. 所有 Swarm 节点的 Linux 系统内核版本不低于 3.16 (在 docker 1.10 后面版本中，已经支持内核 3.10，升级内核实在是一个麻烦事情)
2. 需要一个额外的配置存储服务，例如 Consul、Etcd 或 ZooKeeper
3. 所有的节点都能够正常连接到配置存储服务的 IP 和端口
4. 所有节点运行的 Docker 后台进程需要使用『-cluster-store』和『-cluster-advertise』参数指定所使用的配置存储服务地址

### macvlan

macvlan 对 kernel 版本依赖：Linux kernel v3.9–3.19 and 4.0+。macvlan 允许你在主机的一个网络接口上配置多个虚拟的网络接口，这些网络 interface 有自己独立的 mac 地址，也可以配置上 ip 地址进行通信。macvlan 下的虚拟机或者容器网络和主机在同一个网段中，共享同一个广播域。macvlan 和 bridge 比较相似，但因为它省去了 bridge 的存在，所以配置和调试起来比较简单，而且效率也相对高。除此之外，macvlan 自身也完美支持 VLAN。如果希望容器或者虚拟机放在主机相同的网络中，享受已经存在网络栈的各种优势，可以考虑 macvlan。

### flannel

flannel 是 CoreOS 提供用于解决 Docker 集群跨主机通讯的覆盖网络工具。它的主要思路是：预先留出一个网段，每个主机使用其中一部分，然后每个容器被分配不同的 ip；让所有的容器认为大家在同一个直连的网络，底层通过 UDP/VxLAN(virtual extensible LAN) 等进行报文的封装和转发<sup>75</sup>。

### weave

其中 Weave 是由 Zett.io 公司开发的，它能够创建一个虚拟网络，用于连接部署在多台主机上的 Docker 容器，这样容器就像被接入了同一个网络交换机，那些使用网络的应用程序不必去配置端口映射和链接等信息。外部设备能够访问 Weave 网络上的应用程序容器所提供的服务，同时已有的内部系统也能够暴露到应用程序容器上。Weave 能够穿透防火墙并运行在部分连接的网络上，另外，Weave 的通信支持加密，所以用户可以从一个不受信任的网络连接到主机。

---

<sup>75</sup>参考来源：<https://www.hi-linux.com/posts/30481.html>

## calico

Project Calico 是纯三层的 SDN 实现，没有使用重载网络，它基于 BGP 协议和 Linux 自己的路由转发机制，不依赖特殊硬件，没有使用 NAT 或 Tunnel 等技术。能够方便的部署在物理服务器、虚拟机（如 OpenStack）或者容器环境下。同时它自带的基于 Iptables 的 ACL 管理组件非常灵活，能够满足比较复杂的安全隔离需求。

## Canal (Flannel + Calico)

### 1.6.12 Git 提交规范

规范的 Git 提交能够提供更多信息，方便迅速定位问题、排查问题与回退，还可以自动生成修改文档 (Change log)。提交时定义的提交类型如下：

- feature: 新添加了功能
- refactor: 代码进行了重构
- bugfix: 修复了 Bug，需要带上 bug 编号
- bug: 一般的问题修复
- docs: 添加了文档描述内容
- style: 代码风格调整
- perf: 性能优化
- auto: 脚本自动提交
- chore: 构建过程或辅助工具的变动 (产品外修改，对产品本身没有任何直接影响)

在 Terminal 中提交命令示例：

```
1 git commit -m "[refactor] 移除不应受版本控制但已经提交的文件"
```

为了避免出现不符合规范的 commit，可以添加客户端钩子 (.git/hooks/commit-msg)。使用该钩子来读取作为第一个参数传递的提交信息，然后与规定的格式作比较，就可以使 Git 在提交信息格式不对的情况下拒绝提交：

```
1 #!/usr/bin/env ruby
2 message_file = ARGV[0]
3 message = File.read(message_file)
4 $regex = /\[bugfix|feature|refactor|docs|style|perf: \]/
5 if !$regex.match(message)
```

```
6 puts "[POLICY] Your message is not formatted correctly"
7 exit 1
8 end
```

### 1.6.13 Git 常见问题

用 git 提交时报错：error: RPC failed; HTTP 413 curl 22 The requested URL returned error: 413 Request Entity。调整为 ssh 提交即可：

```
1 git remote set-url origin ssh://xxx@github.org/hello/hello.git
```

### 1.6.14 JVM 的垃圾收集器有几种

JVM 的垃圾收集器一共有 7 种。新生代 3 种 (Serial/ParNew(Parallel New)/Parallel Scavenge)，老年代 3 种 (CMS/Serail Old(MSC<sup>76</sup>)/Parallel Old)，外加年轻代和老年代都会用到的 G1(Garbage First) 收集器。

#### 年轻代收集器 (Young Generation Collector)

在用户的桌面应用场景中，分配给虚拟机管理的内存一般不会很大，收集几十兆甚至一两百兆的新生代，停顿时间完全可以控制在几十毫秒最多一百毫秒以内，只要不频繁发生，这点停顿时间可以接受。所以，Serial 收集器对于运行在 Client 模式下的虚拟机来说是一个很好的选择，可以通过-XX:+UseSerialGC 来强制指定。

Parallel Scavenge<sup>77</sup>收集器除了会显而易见地提供可以精确控制吞吐量的参数，还提供了一个参数-XX:+UseAdaptiveSizePolicy，这是一个开关参数，打开参数后，就不需要手工指定新生代的大小 (-Xmn)、Eden 和 Survivor 区的比例 (-XX:SurvivorRatio)、晋升老年代对象年龄 (-XX:PretenureSizeThreshold) 等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，这种方式称为 GC 自适应的调节策略 (GC Ergonomics)。自适应调节策略也是 Parallel Scavenge 收集器与 ParNew 收集器的一个重要区别。

<sup>76</sup>Mark Sweep Collection 的缩写，不过未找到可信的出处。

<sup>77</sup>Scavenge 有清除的意思。

## 老年代收集器

Serial Old 是 Serial 收集器的老年代版本，它同样是一个单线程收集器，使用“标记-整理”(Mark-Compact) 算法。此收集器的主要意义也是在于给 Client 模式下的虚拟机使用。如果在 Server 模式下，它还有两大用途：在 JDK1.5 以及之前版本 (Parallel Old 诞生以前) 中与 Parallel Scavenge 收集器搭配使用。作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 时使用。它的工作流程与 Serial 收集器相同。

Parallel Old 收集器是 Parallel Scavenge 收集器的老年代版本，使用多线程和“标记-整理”算法。前面已经提到过，这个收集器是在 JDK 1.6 中才开始提供的，在此之前，如果新生代选择了 Parallel Scavenge 收集器，老年代除了 Serial Old 以外别无选择，所以在 Parallel Old 诞生以后，“吞吐量优先”收集器终于有了比较名副其实的应用组合，在注重吞吐量以及 CPU 资源敏感的场所，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。Parallel Old 收集器的工作流程与 Parallel Scavenge 相同。

CMS (Concurrent Mark Sweep) 收集器是一种以获取最短回收停顿时间为目标的收集器，它非常符合那些集中在互联网站或者 B/S 系统的服务端上的 Java 应用，这些应用都非常重视服务的响应速度。从名字上 (“Mark Sweep”) 就可以看出它是基于“标记-清除”算法实现的。CMS 收集器工作的整个流程分为以下 4 个步骤：

初始标记 (CMS initial mark)：仅仅是标记一下 GC Roots 能直接关联到的对象，速度很快，需要“Stop The World”。并发标记 (CMS concurrent mark)：进行 GC Roots Tracing 的过程，在整个过程中耗时最长。重新标记 (CMS remark)：为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。此阶段也需要“Stop The World”。并发清除 (CMS concurrent sweep) 由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS 收集器的内存回收过程是与用户线程一起并发执行的。

G1 (Garbage-First) 收集器是当今收集器技术发展最前沿的成果之一，它是一款面向服务端应用的垃圾收集器，HotSpot 开发团队赋予它的使命是 (在比较长期的) 未来可以替换掉 JDK 1.5 中发布的 CMS 收集器。与其他 GC 收集器相比，G1 具备如下特点：并行与并发 G1 能充分利用多 CPU、多核环境下的硬件优势，使用多个 CPU 来缩短“Stop The World”停顿时间，部分其他收集器原本需要停顿 Java 线程执行的 GC 动作，G1 收集器仍然可以通过并发的方式让 Java 程序继续执行。分代收集与其他收集器一样，分代概念在 G1 中依然得以保留。虽然 G1 可以不需要其他收集器配合就能独立管理整个 GC 堆，但它能够采用不同方式去处理新创建的对象和已存活一段时间、熬过多次 GC 的旧对象来获取更好的收集效果。空间整合 G1 从整体来看是基于

“标记-整理”算法实现的收集器，从局部（两个 Region 之间）上来看是基于“复制”算法实现的。这意味着 G1 运行期间不会产生内存空间碎片，收集后能提供规整的可用内存。此特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次 GC。可预测的停顿这是 G1 相对 CMS 的一大优势，降低停顿时间是 G1 和 CMS 共同的关注点，但 G1 除了降低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在 GC 上的时间不得超过 N 毫秒，这几乎已经是实时 Java (RTSJ) 的垃圾收集器的特征了。

### 1.6.15 控制反转 (IoC)

IoC(Inversion of Control) 的概念是 Michael Mattson 在 1996 年一篇讨论面向对象框架 (Object Oriented Frameworks) 的文章中提出的<sup>78</sup>。在没有 IoC 时，A 对象需要依赖 B 对象时，一般通过在 A 类中通过 new 关键字来创建 B 的实例。A 对 B 形成一种强依赖关系，此时创建 B 对控制权在 A 手上。引入 IoC 容器后，当 A 需要使用 B 时，由容器去实例化 B 并注入到 A。这种由 A 掌握控制权并主动初始化 B 变为由容器初始化并注入到 A 的行为，称之为控制反转 (Inversion of Control)。

#### IoC 的优点

IoC 最大的优点是解耦，对象之间不再有强依赖关系<sup>79</sup>，从而提高了程序的可扩展性 (Extensibility)、可维护性 (Maintainability)。

### 1.6.16 Maven 常见问题解决方案

#### dependencyManagement

在项目的 parent 层，可以通过 dependencyManagement 元素来管理 jar 包的版本，让子项目中引用一个依赖而不用显示的列出版本号。在使用 dependencies 时，父项目声明后，子项目默认自动引入，这样就会造成一个问题，子项目会引入一些根本不需要的依赖包，dependencyManagement 就能够解决此问题，如果子项目需要引用，需要显示的指定，但是可以省略版本号。没有指定具体版本时，从父项目中继承该项，并且 version 和 scope 都读取自父 pom；另外如果子项目中指定了版本号，那么会使用子项目中指定的 jar 版本。

---

<sup>78</sup>论文下载地址：[https://www.researchgate.net/publication/2238535\\_Object-Oriented\\_Frameworks](https://www.researchgate.net/publication/2238535_Object-Oriented_Frameworks)

<sup>79</sup><https://www.zhihu.com/question/23277575>



### provided, compile, runtime, test, system

默认就是 `compile`，什么都不配置也就是意味着 `compile`。`compile` 表示被依赖项目需要参与当前项目的编译，当然后续测试、运行周期也参与其中，是一个比较强的依赖。打包的时候通常需要包含进去。`runtime` 表示被依赖项目无需参与项目的编译，不过后期的测试和运行周期需要其参与。与 `compile` 相比，跳过编译而已，说实话在终端的项目（非开源，企业内部系统）中，和 `compile` 区别不是很大。从参与度来说，`system` 也与 `provided` 相同，不过被依赖项不会从 `maven` 仓库抓，而是从本地文件系统拿，一定需要配合 `systemPath` 属性使用。

### 重新下载 Jar 包

在使用 `Maven` 过程中，可能是由于网络原因，经常会遇到 `jar` 包下载失败的情况（`Maven` 依赖中有 `jar` 包但是 `jar` 包无内容），使用如下命令重新下载未成功下载的 `jar` 包：

```
mvn clean install -e -U
```

`U` 参数强制让 `Maven` 检查所有 `SNAPSHOT` 依赖更新，确保集成基于最新的状态，如果没有该参数，`Maven` 默认以天为单位检查更新。在 `Maven` 执行部署动作时，提示错误：`I/O exception (maven.wagon.providers.http.httpclient.NoHttpResponseException) caught when processing request: The target server failed to respond`。经过检查，是由于 `Mac` 系统设置了默认代理，代理失效后导致 `Maven` 无法部署 `jar` 包，将代理暂时禁用或者解决代理地址失效问题即可修复。

### 查看 Jar 包依赖

从 `apollo client 1.1.0` 升级到 `apollo client 1.2.0` 的过程中，虽然调整了公共依赖的版本为 `1.2.0`，依其他依赖公共项目的模块始终使用的是 `1.1.0` 版本。最后经过一番折腾，终于调整为 `1.2.0` 版本，最大可能的原因是 `parent` 项目中定义了 `1.1.0` 版本（经过排查的确是位于在 `parent` 包中指定了 `1.1.0` 版本的依赖）。`Maven` 依赖的规则之一是路径最近者优先，例如项目 `A` 有如下的依赖关系：`A->B->C->X(1.0)` 和 `A->D->X(2.0)`，则该例子中，`X` 的版本是 `2.0`。规则之二是路径相等，先声明者优先，例如项目 `A` 有如下的依赖关系：`A->B->Y(1.0)` 和 `A->C->Y(2.0)`，若 `pom` 文件中 `B` 的依赖坐标先于 `C` 进行声明，则最终 `Y` 的版本为 `1.0`。但是从解决问题的过程中明白学习到了查看 `Maven` 依赖的命令：

```
1 mvn dependency:tree -Dverbose -Dincludes=com.ctrip.framework.  
    apollo:apollo-client
```

以上命令查看 apollo-client 的依赖结构, 可以看到是哪个模块依赖了 apollo-client, 模块与模块之间的依赖关系, 最终可以找到在多模块项目哪里定义的 apollo-client 依赖。在升级 jar 包的过程中, 遇到依赖找不到的情况, 提示 Dependency 'de.javakaffee:kryo-serializers:0.45' not found。一说是此依赖类型是 bundle 类型<sup>80</sup>, 需要下载插件, 但是最终插件下载也是失败的 (有点像是 IntelliJ Idea 的 Bug)。还有一说是需要更新 Maven Repo 索引, 截止 2019 年 8 月 5 日 Maven 仓库 <https://repo1.maven.org/maven2> 的索引大小为 720MB 左右。Maven 中心仓库的配置:

```
1 <repository>  
2   <id>central</id>  
3   <name>central repository</name>  
4   <url>https://repo1.maven.org/maven2</url>  
5   <layout>default</layout>  
6   <releases>  
7     <enabled>true</enabled>  
8     <updatePolicy>never</updatePolicy>  
9   </releases>  
10  <snapshots>  
11    <enabled>true</enabled>  
12    <updatePolicy>never</updatePolicy>  
13  </snapshots>  
14 </repository>
```

### IntelliJ Idea 下设置 Maven 路径

在 IntelliJ Idea 下设置 Maven 路径为: /usr/local/Cellar/maven@3.5/3.5.4/libexec, 不要直接使用 /usr/local/Cellar/maven@3.5/3.5.4 路径, 否则在 IDE 中执行 Maven 指令时会提示错误: No valid Maven installation found. Either set the home directory in the configuration dialog or set the M2\_HOME environment variable on your system。或者在填写路

<sup>80</sup><https://stackoverflow.com/questions/57346871/maven-dependency-de-javakaffee-kryo-serializers-0-45-not-found-in-intellij-ide>

径时提示 invalid maven home directory。

### 1.6.17 Mac 使用技巧

搜索文件重建 Spotlight 索引，可以搜索整个磁盘上的文件。

```
1 # 开启索引
2 sudo mdutil -a -i on
3 # 索引目录
4 sudo mdutil -E "/"
```

执行命令后即可看到索引的进度。



Figure 1.16: Spotlight 建立索引

**快速跳转目录。**在终端中切换到层级比较深的路径时，需要多次输入 `cd` 命令，或者使用 `tab` 快捷键来自动补全命令，其实终端中快速进行目录切换可以使用 `autojump` 命令，在安装后 `autojump` 后，`cd` 到达过的路径都会加入到 `autojump` 的历史列表里，历史列表会记录所有目录的权重，经常进入的目录权重较高，在快速切换时，就可以使用 `j` 命令直接快速进入某个目录，比如 `j Downloads -> cd /Downloads`。输入如下命令安装 `autojump`：

```
1 # Mac下安装autojump
2 brew install autojump
```

安装完毕后按照提示在 `.bashrc` 文件中添加命令使 `autojump` 生效：

```
1 # .bashrc中添加命令行
```

```
2 [ -f /usr/local/etc/profile.d/autojump.sh ] && . /usr/local/etc/  
    profile.d/autojump.sh  
3 # 使autojump生效  
4 source ~/.bashrc
```

autojump 会在你每次启动命令时记录你当前位置，并把它添加进它自身的数据库中，数据文件的位置在/Users/dolphin/Library/autojump/autojump.txt。

**查看 App 密码。**在使用软件 Sequel Pro 过程中，一段时间之后忘记了数据库密码，软件中显示时默认隐藏了密码，以星号显示。此时可以在钥匙串访问 (Keychain Access) 应用中查看应用密码 (application password)。“钥匙串访问”是一款 macOS 应用，可以储存您的密码和帐户信息，并减少必须记住和管理的密码数量<sup>81</sup>。

**按 Tab 自动提示时，忽略大小写。**Mac 下智能提示默认大小写敏感，例如 Downloads 文件夹在输入小写的 d 按 Tab 时，是不会自动补全的。在 home 目录下新建.inputrc<sup>82</sup>文件，在文件末尾添加如下内容：

```
1 set completion-ignore-case on  
2 set show-all-if-ambiguous on  
3 TAB: menu-complete
```

添加完毕后，执行 source 命令使.inputrc 文件生效，启动新的 Terminal 窗口，输入对应文件夹首字母，按下 Tab 命令自动补全，此时不需要切换大小写也可出现智能提示，非常方便。

**避免 SSH 自动断开链接。**SSH 登陆一段时间后会提示，packet\_write\_wait: Connection to 192.168.1.1 port 22: Broken pipe。在客户端的.ssh/config 文件中添加配置项 ServerAliveInterval，如下代码片段所示：

```
1 Host Demo  
2     UseKeyChain yes
```

<sup>81</sup>来源链接：<https://support.apple.com/zh-cn/guide/keychain-access/kyca1083/mac>

<sup>82</sup>rc 表示 run commands, [Unix: from runcom files on the CTSS system 1962-63, via the startup script /etc/rc] Script file containing startup instructions for an application program (or an entire operating system), usually a text file containing commands of the sort that might have been invoked manually once the system was running but are to be executed automatically each time the system starts up. 参考来源：<https://unix.stackexchange.com/questions/3467/what-does-rc-in-bashrc-stand-for>

```
3 AddKeysToAgent yes
4 IdentityFile ~/.ssh/id_rsa
5 HostName 192.168.1.1
6 ServerAliveInterval 30
7 User demo
```

ServerAliveInterval 表示 ssh 客户端每隔 30 秒给远程主机发送一个 no-op(No Operation) 包, no-op 是无任何操作的意思, 这样远程主机就不会关闭这个 SSH 会话。

终端根据文件类型显示颜色。使用 Mac 时, 比较头疼的是终端颜色默认是白加黑模式。要使 Mac 终端显示颜色, 在 home 目录下的 bash\_profile 文件中添加如下指令:

```
1 export CLICOLOR=1
2 export LSCOLORS=ExGxFxdacxDaDahbadeche
```

LSCOLORS= 后, 共 22 个字母, 每个字母对应一种颜色。2 个字母为一组, 共 11 组。每一组代表一种文件类型。11 组文件类型的含义如下:

- 1 1. directory
- 2 2. symbolic link
- 3 3. socket
- 4 4. pipe
- 5 5. executable (可执行文件, x 权限)
- 6 6. block special
- 7 7. character special
- 8 8. executable with setuid bit set (setuid=Set User ID, 属主身份)
- 9 9. executable without setgid bit set
- 10 10. directory writable to others, with sticky bit
- 11 11. directory writable to others, without sticky bit

一般 Linux 发行版的颜色定义如表所示:

Mac 自带的 Terminal 终端没有文件名高亮和自动提示等功能, 自己折腾还是相当麻烦的, 而且界面不是很好看, 所以安装了 iTerm, 但是 iTerm 也没有达到理想中的美化效果, 网络上对 oh-my-zsh 都是一片赞誉, 截至 2019 年 7 月 24 日, oh-my-zsh 在 Github 上已经有 9 万多星。

Table 1.10: Linux 终端显示颜色定义

文件类型	颜色	用途
压缩文件	红色	
目录文件	蓝色	
块设备文件	黄褐色	
字符设备	黄色	

从 bash 切换到 oh my zsh

bash 是 Mac OS 系统默认的 shell，默认没有 Tab 智能提示时大小写忽略的特性，提示也不够智能，比如默认没有 ssh Tab 主机名自动提示，颇为不便。比 bash 定制性更好的 shell 是 zsh，但是 zsh 的缺点是配置过于复杂，对新手不够友好。oh my zsh 很好的解决了配置繁琐复杂这一痛点，输入如下命令安装 oh my zsh：

```
1 sh -c "$(curl -fsSL https://raw.githubusercontent.com/robbyrussell/oh-my-zsh/master/tools/install.sh)"
```

美化 oh my zsh

安装好 oh my zsh 之后，在配置文件.zshrc 中将主题调整为 amuse：

```
1 ZSH_THEME="amuse"
```

终端打开文件夹

当在终端中切换到目录/Users/dolphin/Library/Caches/IntelliJidea2019.1/时，发现在文件浏览器中根本没有这个目录 (可能 Library 目录是符号链接)。所以此时可以选择在终端中直接打开文件夹：

```
1 # 打开当前所在的文件夹
2 open .
3 # 打开家目录
4 open ~
```

```
5 # 打开家目录下的 Downloads 文件夹
6 open ~/Downloads/
```

### 1.6.18 Apollo 服务端 Docker 部署

在包含 Apollo 的 Dockerfile 的路径 (如 admin-service 的路径: /opt/dabai/app/config-center/docker-apollo/apollo-adminservice) 下执行命令构建 Apollo 的 docker 镜像:

```
1 # 打包调整好配置的 Apollo 文件
2 # 具体的配置根据不同环境而定
3 # 一般情况下调整数据库链接即可
4 # 用于构建镜像
5 zip -q -r apollo-adminservice-1.4.0-github.zip apollo-adminservice
   -1.4.0-github
6 # 构建 Apollo 测试配置服务测试环境镜像
7 sudo docker build -t apollo-adminservice-uat:latest .
```

t 参数表示镜像的名字或者标签 (tag), 这里镜像的名字是 apollo-adminservice, 指定 tag 是 latest, 格式通常 name:tag 或者 name 格式, 不指定 tag 时默认是 latest, 最后的参数点表示当前路径。运行如下命令启动容器:

```
1 # 运行测试环境镜像 (暂未包含 volume 映射)
2 sudo docker run -p 18083:8090 -d --name apollo-adminservice apollo
   -adminservice
3 # 启动镜像, 并映射日志文件路径
4 sudo docker run -d -p 18089:8090 -v /opt/logs:/opt/logs \
5 --name apollo-adminservice-uat apollo-adminservice-uat
6 sudo docker run -d -p 18090:8080 -v /opt/logs:/opt/logs \
7 --name apollo-configservice-uat apollo-configservice-uat
```

在启动容器的时候有一个细节, 可以将容器中的日志文件路径映射到本地磁盘。启动时有可能 Docker 容器启动成功, 但是容器中的应用可能启动失败, 若需要在容器停止后分析日志, 可以在提前映射的卷下找到。否则使用 docker logs 命令可以看到

成功的日志，但同时无法登陆已经停止的容器查看应用启动日志。将容器中的运行目录拷贝到本地磁盘：

```
1 sudo docker cp b84f9ccac7d4:/apollo-adminservice /opt/dabai/app/  
    docker/apollo/apollo-adminservice
```

新增卷映射，这里暂时还不知道如何动态调整卷映射，采用的方式是停止和移除原来的容器后，重新创建容器时新增映射卷 (Volumne)：

```
1 # 运行配置服务测试环境镜像  
2 sudo docker run -p 18089:8090 -d --name apollo-adminservice -v /  
    opt/dabai/app/docker/apollo/apollo-adminservice:/apollo-  
    adminservice apollo-adminservice  
3 sudo docker run -p 18089:8090 -d --name apollo-adminservice-uat \  
4 -v /opt/dabai/app/docker/apollo/apollo-adminservice-uat:/apollo-  
    adminservice \  
5 -v /opt/logs:/opt/logs apollo-adminservice-uat
```

### Apollo 客户端无法读取内网配置

需要注意的是，在部署完毕 Apollo 后，客户端启动应用获取服务端配置时，客户端默认还是请求的服务端内网的地址（地址是由服务端返回给客户端）。例如启动客户端时会提示连接超时警告：

```
1 2019-07-10 12:40:27.221 WARN 55483 --- [          main] c.c.f.a.  
    i.AbstractConfigRepository      : Sync config failed,  
    will retry. Repository class com.ctrip.framework.apollo.  
    internals.RemoteConfigRepository, reason: Load Apollo  
    Config failed - appId: 0010020001, cluster: default,  
    namespace: application, url: http://172.17.0.10:8080/  
    configs/0010020001/default/application?ip=192.168.50.25 [  
    Cause: Could not complete get operation [Cause: Read timed  
    out]]
```



可以发现，客户端此时是从地址 172.17.0.10 来读取的配置，而 172.17.0.10 是公网阿里云主机上 docker 容器的内部地址，当前 PC 所处的局域网主机默认是无法直接访问的。要让当前局域网的主机读取 Apollo 公网暴露的映射地址，需要调整 Apollo 容器应用的启动命令，手动指定公网映射地址。在 Apollo ConfigService 的启动脚本 (startup.sh) 中指定远程连接 URL：

```
1 export JAVA_OPTS="$JAVA_OPTS -XX:ParallelGCThreads=4 -  
    XX:MaxTenuringThreshold=9 -XX:+DisableExplicitGC -XX:+  
    ScavengeBeforeFullGC -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+  
    ExplicitGCInvokesConcurrent -XX:+  
    HeapDumpOnOutOfMemoryError -XX:-OmitStackTraceInFastThrow  
    -Duser.timezone=Asia/Shanghai -Dclient.encoding.override=  
    UTF-8 -Dfile.encoding=UTF-8 -Djava.security.egd=file:/dev  
    /./urandom -Deureka.instance.homePageUrl=http://dolphin.  
    example.com:18087"
```

其中 Deureka.instance.homePageUrl 参数是手动添加的，服务端返回地址时，就会返回此手动配置的地址，而此处指定的配置的公网地址，处于局域网的客户端就可以获取到了，但是一般情况下不建议将此地址暴露出去，客户端和服务端部署在同一个局域网即可。

### 1.6.19 Apollo 客户端使用

Apollo（阿波罗）是携程框架部门研发的分布式配置中心，能够集中化管理应用不同环境、不同集群的配置，配置修改后能够实时推送到应用端，并且具备规范的权限、流程治理等特性，适用于微服务配置管理场景<sup>83</sup>。客户端的配置与使用步骤如下，首先在客户端主机/opt/settings 路径配置 server.properties。

```
1 # 指定环境（可选项）  
2 env=PRO  
3 # Apollo Meta Server(必选项)  
4 apollo.meta=http://{external-ip}:8082  
5 # PRO 环境配置地址（必选项）
```

<sup>83</sup><https://github.com/ctripcorp/apollo>

```
6 pro.meta=http://{external-ip}:8082
```

需要注意的的点是 `external-ip` 需要是公网的 IP，如果是内网 IP 那么客户端要能够直接访问元数据服务器 (Apollo Meta Server)。其次在应用的 `application.properties` 文件指定 `app.id`。

```
1 # 指定应用 appId(必选项)
2 app.id=0000000000
```

`appId` 可以根据实际的情况指定 (有多种不同优先级的指定方式，具体参考官方文档)，例如用产品线 (003) + 项目名称 (004) + 应用名称 (0005) 的模式来定义，定义好的 `appId` 就是 0030040005。新增获取配置的实体 `TestJavaConfigBean`：

```
1 public class TestJavaConfigBean {
2     @Value("${timeout:100}")
3     private int timeout;
4     private int batch;
5
6     @Value("${batch:200}")
7     public void setBatch(int batch) {
8         System.out.println("TestJavaConfigBean:" + batch);
9         this.batch = batch;
10    }
11
12    public int getTimeout() {
13        return timeout;
14    }
15
16    public int getBatch() {
17        return batch;
18    }
19 }
```

新增 `AppConfig`：

```
1 @Configuration
2 @EnableApolloConfig
3 public class AppConfig {
4     @Bean
5     public TestJavaConfigBean javaConfigBean() {
6         return new TestJavaConfigBean();
7     }
8 }
```

同时应用入口处添加 `EnableApolloConfig` 注解。配置完毕后，启动应用，可以发现读取的 `timeout` 字段的配置与服务器的配置一致，修改服务器的配置，客户端会通过 `long polling` 实时更新项目配置项。配置项在 `Linux/Unix` 类型操作系统缓存在路径 `/opt/data/0000000000/config-cache` 下，缓存文件采用的命名模式为应用 ID 后跟集群名称，最后是命名空间，例如：

0010010003+default+TEST1.EUREKA.properties

0010010003 是应用的 ID，`default` 是集群名称，`TEST1.EUREKA` 是命名空间名称。在启动时，如果遇到提示：

```
1 Sync config failed, will retry. Repository class com.ctrip.
   framework.apollo.internals.RemoteConfigRepository, reason:
   Get config services failed from http://172.10.175.21
   :18082/services/config?appId=0010010001\&ip=192.168.50.25
   [Cause: Could not complete get operation [Cause: Read
   timed out]]
```

而将链接拷贝到浏览器却可以正常访问，此时检查电脑是否设置了代理。一种是可以暂时去掉系统代理设置，一种是可以开发工具如 `IntelliJ Idea` 中设置代理。

## 1.7 05 月

### 1.7.1 Docker 常用命令

查看容器资源占用情况：

```
1 docker stats --no-stream
```

`no-stream` 表示不用定时刷新，只输出容器当前状态。查看容器所用 Linux 版本（注意不是宿主机 Linux 版本）：

```
1 cat /etc/issue
```

查看容器的配置信息：

```
1 # Apollo portal fe6118c73539
2 docker inspect {container_id}
3 cd /opt/dabai/app/docker/apollo/apollo-portal/apollo-portal
```

将容器中的文件拷贝到本地：

```
1 docker cp 12ba0764beff:/apollo-portal /opt/dabai/app/docker/apollo
  /apollo-portal
```

登陆到 Docker 容器：

```
1 docker exec -i -t fbf68c2e3e27 /bin/bash
```

移除异常状态的 Docker 进程：

```
1 # Docker 1.13 版本以后
2 # 可以使用 docker containers prune 命令，删除孤立的容器。
3 sudo docker container prune
```

移除容器：

```
1 # Docker 移除容器
2 # 添加 f(force) 参数强制移除
3 docker rm ae5b02152e0f
```

停止命名以 `rancher` 开头的容器：

```
1 docker stop $(docker ps -a | grep "rancher*" | awk '{print $1}')
```

### 1.7.2 Docker 数据卷

Docker 镜像是由多个文件系统（只读层）叠加而成。当启动一个容器的时候，Docker 会加载只读镜像层并在镜像栈顶部添加一个读写层。如果运行中的容器修改了现有的一个已经存在的文件，那该文件将会从读写层下面的只读层复制到读写层，该文件的只读版本仍然存在，只是已经被读写层中该文件的副本所隐藏。当删除 Docker 容器，并通过该镜像重新启动时，之前的更改将会丢失。在 Docker 中，只读层及在顶部的读写层的组合被称为 **Union File System**（联合文件系统）<sup>84</sup>。数据卷（Data Volumes）是存在于一个或多个容器中的特定文件或文件夹，这个文件或文件夹以独立于 Docker 文件系统的形式存在于宿主机中，所以可以将容器以及容器产生的数据分离开来。数据卷的最大特点是：其生存周期独立于容器的生存周期<sup>85</sup>。数据卷主要解决了以下问题：

- 不能在宿主机上很方便地访问容器中的文件
- 无法在多个容器之间共享数据
- 当容器删除时，容器中产生的数据将丢失

以下代码片段示例创建一个数据卷映射：

```
1 sudo docker run -p 18084:8070 \  
2 --name apollo-portal \  
3 -v /opt/apollo-portal-1.4.0-github:/apollo-portal -d ce518631ced
```

`docker run` 命令先是利用镜像创建了一个容器，然后运行这个容器。实际上，`docker run` 就是 `docker create` 和 `docker start` 两个命令的组合。`/opt/apollo-portal-1.4.0-github` 是磁盘上的文件夹，`/apollo-portal` 是容器内的文件夹。Docker 挂载数据卷的默认权限是可读写（rw），用户也可以通过 `ro` 标记指定为只读：

<sup>84</sup><http://dockone.io/article/128>

<sup>85</sup><https://www.cnblogs.com/sparkdev/p/8504050.html>

```
1 sudo docker run -p 18084:8070 \  
2 --name apollo-portal \  
3 -v /opt/apollo-portal-1.4.0-github:/apollo-portal:ro \  
4 -d ce518631ced
```

加了:ro(Read Only) 标记之后, 容器内挂载的数据卷内的数据就变成只读的了。

### 1.7.3 Java 中的 volatile 关键字

volatile 的两大特性**禁止重排序 (Happens-Before Guarantee)**(编译器的重排序和 CPU 指令的重排序)、**可见性 (Visibility Guarantee)**, 例如, JVM 或者 JIT 为了获得更好的性能会对语句重排序, 但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与其他语句重排序<sup>86</sup>。

#### 禁止重排序 (Happens-Before Guarantee)

为了提高程序运行性能, 一般会对程序运行的指令重排 (instruction reordering)。指令重排序 (instruction reordering) 的概念大约出现在 Pentium Pro 系列上。例如 CPU 需要执行如下代码片段:

```
1 a = b + c;  
2 d = e - f;
```

在执行 b 加 c 的时候, 先决条件是变量 b 和 c 都已经被读取进入到寄存器。但有时执行到 add 操作时, 可能变量 b 和 c 还没有装载完毕, 此时 CPU 就必须停顿等待, 后面执行的指令也会依次受到影响。指令重排序就是在编译器层面和 CPU 运行时层面不一定严格按照编写代码的顺序执行, 可以是先执行 e 和 f 的装载, 执行完毕后 b 和 c 有更大概率已经装载完毕, 那么此时 CPU 就可以在没有任何停顿的情况下执行 add 操作, 从而避免停顿提高了指令执行效率<sup>87</sup>。如下代码片段可以体会指令重排序对多线程应用的影响:

```
1 static int x, y, m, n; //测试用的信号变量  
2
```

<sup>86</sup>文章参考: <https://my.oschina.net/tantexian/blog/808032>

<sup>87</sup>来源文章: <https://redspider.gitbook.io/concurrent/di-er-pian-yuan-li-pian/7>

```
3 public static void main(String[] args) {
4     int count = 500000;
5     for (int i = 0; i < count; ++i) {
6         x = y = m = n = 0;
7         //线程一
8         Thread one = new Thread() {
9             @Override
10            public void run() {
11                m = 1;
12                x = n;
13            };
14        };
15        //线程二
16        Thread two = new Thread() {
17            @Override
18            public void run() {
19                n = 1;
20                y = m;
21            };
22        };
23        //启动并等待线程执行结束
24        one.start();
25        two.start();
26        one.join();
27        two.join();
28        //输出结果
29        if (x == 0 && y == 0) {
30            System.out.println("index:" + i + " {x:" + x + ",y:" +
31                y + "}");
32        }
33        System.out.println("complete!");
34    }
```

如上代码片段，当  $x$  为 0 且  $y$  为 0 的时候，输出索引位置，每次运行触发的索引的位置可能变化，在一定循环范围内发生的次数也可能不同，不同硬件平台可能也会有差异。在 Mac Book Air 2015 平台上反复运行多次，循环次数在十万级别才会触发。 $x$  和  $y$  同时为 0，说明代码并不是严格按照顺序执行，发生了指令重排 (instruction reordering)。volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。如果把加入 volatile 关键字的代码和未加入 volatile 关键字的代码都生成汇编代码，会发现加入 volatile 关键字的代码会多出一个 lock 前缀指令。lock 前缀指令实际相当于一个内存屏障 (Memory Barrier<sup>88</sup>)，内存屏障提供了以下功能：

1. 重排序时不能把后面的指令重排序到内存屏障之前的位置
2. 使得本 CPU 的 Cache 写入内存
3. 写入动作也会引起别的 CPU 或者别的内核无效化其 Cache，相当于让新写入的值对别的线程可见

### volatile 与原子性

某些情况下，volatile 还能提供原子性，如读 64 位数据类型，long 和 double 都不是原子的，但 volatile 类型的 double 和 long 就是原子的。如下命令根据简单的示例代码编译出 ASM 代码：

```
1 java -server -Xcomp -XX:+UnlockDiagnosticVMOptions -XX:-Inline \  
2 -XX:TieredStopAtLevel=1 \  
3 -XX:CompileCommand=print,*Test.main Test > Test.asm
```

-Xcomp 表示永远以编译模式运行，-XX:-Inline：禁止内联优化。添加 TieredStopAtLevel 参数仅仅输出一段 main 函数汇编代码<sup>89</sup>，TieredCompilation 打开“多层编译” (tiered compilation)。在该模式下，代码会先被解释器执行，积累到足够热度的时候由 client compiler (C1) 编译，然后继续积累热度到一定程度会进一步被 server compiler (C2) 重新以更高的优化程度编译<sup>90</sup>。CompileCommand 以及后面带的参数表示仅仅输出 Test 类的 main 方法的汇编代码。根据指令获得的汇编代码片段如下：

<sup>88</sup>[https://en.wikipedia.org/wiki/Memory\\_barrier](https://en.wikipedia.org/wiki/Memory_barrier)

<sup>89</sup>具体编译过程：<https://stackoverflow.com/questions/56404146/why-does-java-compile-to-assembly-twice>

<sup>90</sup>来源链接：<https://rednaxelafx.iteye.com/blog/1022095>



```
1 ;*getstatic a
2 0x000000001118487e5: mov     0x68(%rsi),%edi
3
4 0x000000001118487e8: inc     %edi
5 0x000000001118487ea: mov     %edi,0x68(%rsi)
6 ;*putstatic a
7 0x000000001118487ed: lock addl $0x0,(%rsp)
```

从汇编指令可以看出 `a++` 操作实际上是由 (Load、Increment、Store、Memory Barriers) 四个步骤组成，所以由 `volatile` 关键字修饰的 `a++` 操作本身并不是原子性的<sup>91</sup>，在当前线程执行 `inc(increment)` 操作时，如果有其他线程修改了 `edi` 目标索引寄存器<sup>92</sup>(Extended Destination Index)，后一步写入时就会将其他线程的修改覆盖，出现非预期的结果。关于 `lock` 指令产生的内存屏障，不再赘述。

### 可见性 (Visibility Guarantee)

在 JMM(Java Memory Model) 级别，`volatile` 修饰的变量每次回写都会写入到主内存，而当前线程缓存，同样，读取 `volatile` 修饰的变量也会首先从主内存读取，而不是本地缓存<sup>93</sup>。在硬件级别是通过缓存一致性协议来保证。

## 1.7.4 Fluent Bit 日志写入测试

Fluent Bit<sup>94</sup>是一个跨平台的日志搜集处理和转发应用，采用 C 语言编写，非常轻量级，占用的资源较少，相对于 Fluent 性能更优一些。Fluent Bit 更适用于嵌入设备等资源受限的场景。在 Rancher 中安装了 EFK(Elasticsearch, Fluentd, Kibana) 后，登录 Fluent Bit 容器，查看 Fluent Bit 配置文件 (`/fluent-bit/etc/fluent-bit.conf`)：

```
1 [SERVICE]
2     Flush      1
3     Daemon     Off
```

<sup>91</sup>具体原因可以参考 Intel 的官方文档：<https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>

<sup>92</sup>EDI 寄存器可参考：[https://en.wikibooks.org/wiki/X86\\_Assembly/X86\\_Architecture](https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture)

<sup>93</sup>更多内容可以参考《JSR-133: Java Memory Model and Thread Specification》，下载链接：<http://www.cs.umd.edu/~pugh/java/memoryModel/CommunityReview.pdf>

<sup>94</sup><https://fluentbit.io/>

```
4   Log_Level      info
5   Parsers_File   parsers.conf
6
7   [INPUT]
8       Name          tail
9       Path           /var/log/containers/*.log
10      Parser         docker
11      Tag             kube.*
12      Refresh_Interval 5
13      Mem_Buf_Limit   5MB
14      Skip_Long_Lines On
15
16  [FILTER]
17      Name            kubernetes
18      Match            kube.*
19      Kube_URL          https://kubernetes.default.svc:443
20      Kube_CA_File      /var/run/secrets/kubernetes.io/
21                        serviceaccount/ca.crt
22      Kube_Token_File    /var/run/secrets/kubernetes.io/
23                        serviceaccount/token
24
25  [OUTPUT]
26      Name             es
27      Match             *
28      Host              efk-elasticsearch
29      Port              9200
30      Logstash_Format   On
31      Retry_Limit       False
32      Type              flb_type
33      Logstash_Prefix    kubernetes_cluster
```

从 Fluent Bit 容器的配置文件可以看出，在 INPUT 配置中，Fluent Bit 监听容器目录下的日志文件 (/var/log/containers/\*.log) 变化，将内容发送到 ElasticSearch。

### 1.7.5 Rancher 2.x 部署入门

Rancher<sup>95</sup>是一个容器管理平台，由 Rancher Labs 公司开发。Rancher Labs 成立于 2014 年，方向是开发操作系统级别的可视化工具，主要与 Linux 容器相关<sup>96</sup>，主打产品是 Rancher 容器编排平台和 RancherOS。Rancher 容器编排平台目前有 1.x 和 2.x 两个主流版本，这里选择的是 Rancher 2.x 版本。本意是计划用轻量级的 Cattle 管理容器编排 (Container Orchestration)，将开发与运维自动化，奈何 Rancher 2.x 不支持 Cattle，不过经过 Rancher 的简化，部署 Kubernetes 已经不再那么繁琐，就是它了。Rancher 的部署环境如下：

- 操作系统版本：CentOS Linux release 7.6.1810 (Core)
- Rancher 版本：v2.2.3-Stable
- Docker 版本：Docker version 1.13.1, build b2f74b2/1.13.1

首先运行如下命令在节点 A 主机上部署 Rancher，本次部署的 Rancher 2.2.3 稳定版本 (Stable)：

```
1 sudo docker run -d --restart=unless-stopped \  
2     -p 9080:80 \  
3     -p 443:443 rancher/rancher:stable
```

部署完毕后，在浏览器中输入地址 <https://localhost:443> 进入 Rancher 界面，首先新增集群，注意选择添加主机自建 Kubernetes 集群 Custom，由于当前主机资源有限，同时选择 Etcd、Control、Worker 角色，将不同角色的资源部署在一台主机上，角色选择完毕后注意下方的高级选项，在里面填写 Agent 节点的内外网地址和名称。Etcd 角色节点用于保存集群自身的信息，Control 角色的节点用与提供 K8s 需要的 API server 和其他相关组件，Worker 节点承载实际的应用的部署<sup>97</sup>。填写完毕相关信息后，在 Agent 节点上执行如下命令启动 Rancher Agent：

```
1 sudo docker run --privileged --restart=unless-stopped \  
2     --net=host -v /etc/kubernetes:/etc/kubernetes \
```

<sup>95</sup><https://www2.cnrancher.com/>

<sup>96</sup>[https://en.wikipedia.org/wiki/Rancher\\_Labs](https://en.wikipedia.org/wiki/Rancher_Labs)

<sup>97</sup><https://rancher.com/an-introduction-to-rke/>

```
3 -v /var/run:/var/run rancher/rancher-agent:v2.2.3 \  
4 --server https://192.168.1.101 \  
5 --token qjch4bh7s8sn2q78v8f7j \  
6 --ca-checksum f27db6a43229ef71d1798d832cbd860 \  
7 --node-name custom --address 21.16.99.213 \  
8 --internal-address 172.17.48.246 \  
9 --etcd --controlplane --worker
```

需要注意的是, 不同的 Agent 节点需要指定不同的角色, 一般情况下 Etcd 角色和 Control 角色一个节点有即可, 涉及到高可用时再考虑新增。Agent 节点执行命令后, 初始化过程是这样的<sup>98</sup>:

- 每个执行的节点上, 创建一个叫 rancher-agent 的容器
- rancher-agent 容器会继续创建 rke-tools(Rancher Kubernetes Engine Tools) 的容器, 然后用它来初始化容器的宿主机各个组件, 比如 Kubelet, Kube-proxy 等。RKE(Rancher Kubernetes Engine) 的目标是降低 Kubernetes 安装的复杂性
- 在 Rancher 的 UI 界面会同步显示正在进行的步骤的信息, 整个过程无需干预

等待节点创建完毕后, 在 Rancher UI 界面上就可以看到新加入的 Agent 节点了, 主机列表效果如图1.17所示。至此, 简单的部署了 Rancher 并添加了 2 个节点, 目前计划其中一个节点做应用的容器化自动发布, 一个节点用 EFK(ElasticSearch、Fluentd 和 Kiabana) 做一个统一的日志搜集分析中心 (Logging Center)。

### 1.7.6 HashMap 的 put 操作是如何定位元素的

HashMap 基本的数据结构是数组加链表的形式, JDK1.8 又增加了红黑树<sup>99</sup>(Red-black tree) 的实现, 使得在超过一定数量的 hash key 冲突后, 链表结构查找、插入、删除复杂度能够保持在  $O(\log n)$ , 避免复杂度退化为  $O(n)$ 。例如程序执行下面代码:

```
1 map.put("dolphin", "小强");
```

<sup>98</sup><http://code2life.top/2018/10/16/0031-rancher-trial/>

<sup>99</sup><https://zh.wikipedia.org/wiki/>

## 主机列表

<div> <div>暂停</div> <div>驱散</div> <div>删除</div> </div>				
<input type="checkbox"/> 状态	名称	角色	版本	处理器
<input type="checkbox"/> Active	app-node-devo... / 172...	全部	v1.13.5 18.9.6	0.7/4 Cores
<input type="checkbox"/> Active	app-node-efk / 172.1...	Worker	v1.13.5 18.9.6	0.4/4 Cores

Figure 1.17: Rancher Agent 主机列表

系统将调用“dolphin”这个 key 的 hashCode() 方法得到其 hashCode 值，然后再通过后面的扰动运算和与运算来定位该键值对在桶 (Bucket) 上的存储位置。JDK 1.8 获取扰动后 hashCode 的算法如下：

```

1 static final int hash(Object key) {
2     int h;
3     # 扰动hashCode
4     # HashMap数组初始长度较短时降低冲突
5     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >> 16);
6 }

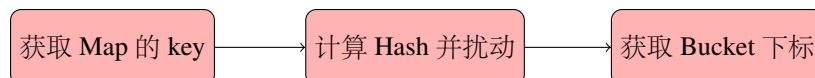
```

高位异或 (XOR) 低 16 位这段代码是扰动函数 (agitate function)，扰动函数的说法目前只在博客上有出现<sup>100</sup>。key.hashCode() 获取到 key 的 hashCode 值，再进行高位运算，通过 hashCode() 的高 16 位异或低 16 位实现，得到目标 hashCode，hashCode 与桶的长度进行与运算 (&) 得到桶的下标。采用这种方式，既减少了系统的开销，也不会造成因为高位没有参与下标的计算 (HashMap 初始数组长度比较小时)，从而引起的更高概率碰撞。

This mixes the high bits of the hash code with the low bits, to improve the randomness of the lower bits. For the case above where there is a high collision rate, there is an improvement.

<sup>100</sup><http://vanillajava.blogspot.com/2015/09/an-introduction-to-optimising-hashing.html>

经过对 HashMap 的 put 原始码分析, HashMap 的 put 方法定位元素流程大致如下:



### 1.7.7 Mac Mojave 编译 OpenJDK 10

编译之前,可以先阅读官方文档<sup>101</sup>,了解编译需要的依赖和编译步骤。基本的步骤包含获取源码,配置,构建,验证构建结果。在 Mac Mojave 下的编译环境:

- 操作系统: Mac OS Mojave 10.14.4 (18E227)
- Boot JDK 版本: java version "1.8.0\_152-ea" Java(TM) SE Runtime Environment (build 1.8.0\_152-ea-b05) Java HotSpot(TM) 64-Bit Server VM (build 25.152-b05, mixed mode)
- C Compiler: Version 9.1.0
- C++ Compiler: Version 9.1.0
- Toolchain: clang
- Xcode 版本: 9.4

Xcode 10 中将 c++ 相关的依赖库由 stdlibc++ 替换成了 libc++, 会影响编译,建议使用 Xcode 9.x 版本。确保系统已安装 freetype 和 ccache:

```
1 brew install freetype ccache
```

确保已经安装分布式版本管理工具 Mercurial:

```
1 # Mac OS 安装 mercurial
2 brew install mercurial
```

<sup>101</sup><https://cr.openjdk.java.net/~ihse/demo-new-build-readme/common/doc/building.html#boot-jdk-requirements>

使用 `hg` 命令获取 OpenJDK 原始码时, `clone` 命令获取可能会失败, 我采取的方式是先在境外的服务器上 `clone` 完毕再采用 `rsync` 命令同步到本机:

```
1 # 获取原始码
2 hg clone http://hg.openjdk.java.net/jdk10/jdk10
3 cd jdk10
4 bash ./get_source.sh
5 # 同步到本地 (增量同步)
6 rsync -avz --progress prod-book-db:/root/source/jdk10 .
```

`hg` 命令是 Mercurial 版本控制系统的命令。Mercurial 是一种轻量级分布式版本控制系统, 采用 Python 语言实现, 易于学习和使用, 扩展性强。其是基于 GNU General Public License (GPL) 授权的开源项目<sup>102</sup>。除了 Mercurial, 分布式版本控制系统领域还有一些其他的系统, 如 GNU arch, monotone, Bazaar, git, darcs。目前已知的版本控制系统比对见维基百科词条<sup>103</sup>。配置参数:

```
1 sh configure --with-debug-level=slowdebug --enable-ccache --
    disable-warnings-as-errors --with-freetype=/usr/local/
    Cellar/freetype/2.10.0
```

在 `configure` 时, 一共可以指定三种级别: `release`, `fastdebug`, `slowdebug`, `slowdebug` 含有最丰富的调试信息, 没有这些信息, 很多执行可能被优化掉, 单步执行时, 可能看不到一些变量的值。所以最好指定 `slowdebug` 为编译级别。

```
1 # 关联到编译时使用的 gcc
2 sudo ln -s /Users/dolphin/Desktop/Xcode.app/Contents/Developer/usr
    /bin/gcc /usr/bin/gcc
3 sudo ln -s /Users/dolphin/Desktop/Xcode.app/Contents/Developer/usr
    /bin/g++ /usr/bin/g++
```

执行编译:

<sup>102</sup><https://www.ibm.com/developerworks/cn/opensource/os-cn-mercurial/index.html>

<sup>103</sup>[https://en.wikipedia.org/wiki/Comparison\\_of\\_version-control\\_software](https://en.wikipedia.org/wiki/Comparison_of_version-control_software)

```
1 make all
```

编译完成后，会输出 Finished building target 'all' in configuration 'macosx-x86\_64-normal-server-slowdebug'。切换到编译生成到目标目录后，使用如下命令验证结果：

```
1 dolphins-MacBook-Air:bin dolphin$ ./java -version
2 openjdk version "10-internal"
3 OpenJDK Runtime Environment (slowdebug build 10-internal+0-adhoc.
   dolphin.jdk10)
4 OpenJDK 64-Bit Server VM (slowdebug build 10-internal+0-adhoc.
   dolphin.jdk10, mixed mode)
```

### 1.7.8 采用离线 Cookie 通过终端命令行下载 Apple 开发工具

在下载 Apple 开发工具 Xcode 时，本机的网络只有 400KB 左右，开发工具 Xcode 本身有 4GB+，完全下载完毕要几个小时。而 Apple 网站下载不支持断点续传，在下载大文件时经常是下载到一半突然就中断，已经下载的部分也无法继续接着下载，只能重新开始下载。刚好手头有国外的云主机，可以在云主机上下载完毕后再采用同步工具离线到本地。由于 Apple 网站在下载之前需要验证用户的 Cookie 信息，直接在终端中通过命令行下载默认是没有用户的 Cookie 信息的，用户的 Cookie 信息缓存在登录过 Apple 网站的浏览器中。首先在 Google Chrome 浏览器上安装 cookies.txt Chrome extension<sup>104</sup>插件，获取到登录 Apple 下载网站后浏览器缓存的用户 Cookie 信息。从浏览器获取到 Cookie 信息后，就可以登录云主机利用 Cookie 文本使用如下命令下载<sup>105</sup>Apple 开发者工具 Xcode：

```
1 wget --load-cookies=cookies.txt https://download.developer.apple.
   com/Developer_Tools/Xcode_8.3.3/Xcode8.3.3.xip
```

在阿里云上使用命令下载速度在 12MB 每秒左右，下载完毕后再通过阿里云上的静态文件服务将 Xcode 文件下载到本地（此时支持断点续传，不用担心中途断开后需

<sup>104</sup><https://chrome.google.com/webstore/detail/cookiestxt/njabckikapfppfapmjgojcnbfjnfjfg?hl=en>

<sup>105</sup><https://stackoverflow.com/questions/4081568/downloading-xcode-with-wget-or-curl>



要重头开始下载的问题)。

```
1 # c 参数支持断点续传
2 wget -c http://{ip}:{port}/path/Xcode8.3.3.xip
```

### 1.7.9 CentOS 编译 OpenJDK 10

编译之前,可以先阅读官方文档<sup>106</sup>,了解编译需要的依赖和编译步骤。基本的步骤包含获取源码,运行配置,执行构建,验证构建结果。编译环境:

- 操作系统: CentOS Linux release 7.6.1810 (Core)
- Boot JDK 版本: openjdk version "1.8.0\_212" OpenJDK Runtime Environment (build 1.8.0\_212-b04) OpenJDK 64-Bit Server VM (build 25.212-b04, mixed mode)
- C Compiler 版本: Version 4.8.5
- C++ Compiler 版本: Version 4.8.5

注意编译环境非常重要,不同的平台,不同的操作系统版本,不同的 GCC 版本,都有可能影响编译。确保系统已安装 freetype 和 ccache:

```
1 yum install freetype ccache -y
```

FreeType 库是一个完全免费 (开源) 的、高质量的且可移植的字体引擎,它提供统一的接口来访问多种字体格式文件,包括 TrueType, OpenType, Type1, CID, CFF, Windows FON/FNT, X11 PCF 等<sup>107</sup>。Ccache 是一个编译工具,可以加速 gcc 对同一个程序的多次编译。确保已经安装 mercurial:

```
1 # CentOS 安装 mercurial
2 yum install mercurial
```

<sup>106</sup><https://cr.openjdk.java.net/~ihse/demo-new-build-readme/common/doc/building.html#boot-jdk-requirements>

<sup>107</sup><https://blog.csdn.net/carson2005/article/details/7221318>

使用 `hg` 命令获取 OpenJDK 原始码, 注意如果物理机器在国内的话, `hg` 命令克隆源码失败的概率很大, 本人采用的方案是在 Google Cloud 上将源码获取 (节点在新加坡, 国外下载速度一般在 5MB-30MB 之间), 再采用 `rsync` 命令将源码同步到需要编译的机器上:

```
1 hg clone http://hg.openjdk.java.net/jdk10/jdk10
2 cd jdk10
3 bash ./get_source.sh
```

`hg` 命令是 Mercurial 版本控制系统的命令。Mercurial 是一种轻量级分布式版本控制系统, 采用 Python 语言实现, 易于学习和使用, 扩展性强。其是基于 GNU General Public License (GPL) 授权的开源项目<sup>108</sup>。除了 Mercurial, 分布式版本控制系统领域还有一些其他的系统, 如 GNU arch, monotone, Bazaar, git, darcs。目前已知的版本控制系统比对应见维基百科词条<sup>109</sup>。配置参数:

```
1 bash configure --with-debug-level=slowdebug --with-jvm-variants=
    server --with-target-bits=64 --enable-ccache --with-memory
    -size=8000
```

在 `configure` 时, 一共可以指定三种级别: `release`, `fastdebug`, `slowdebug`, `slowdebug` 含有最丰富的调试信息, 没有这些信息, 很多执行可能被优化掉, 单步执行时, 可能看不到一些变量的值。所以最好指定 `slowdebug` 为编译级别。配置命令执行成功后, 会输出当前的环境信息:

```
1 A new configuration has been successfully created in
2 /root/source/jdk10/build/linux-x86_64-normal-server-slowdebug
3 using configure arguments '--with-debug-level=slowdebug --with-jvm
    -variants=server --with-target-bits=64 --enable-ccache --
    with-memory-size=8000'.
4
5 Configuration summary:
6 * Debug level:    slowdebug
```

<sup>108</sup><https://www.ibm.com/developerworks/cn/opensource/os-cn-mercurial/index.html>

<sup>109</sup>[https://en.wikipedia.org/wiki/Comparison\\_of\\_version-control\\_software](https://en.wikipedia.org/wiki/Comparison_of_version-control_software)

```
7 * HS debug level: debug
8 * JDK variant:      normal
9 * JVM variants:     server
10 * OpenJDK target: OS: linux, CPU architecture: x86, address
    length: 64
11 * Version string: 10-internal+0-adhoc.root.jdk10 (10-internal)
12
13 Tools summary:
14 * Boot JDK:         openjdk version "1.8.0_212" OpenJDK Runtime
    Environment (build 1.8.0_212-b04) OpenJDK 64-Bit Server VM
    (build 25.212-b04, mixed mode) (at /usr/lib/jvm/java
    -1.8.0-openjdk-1.8.0.212.b04-0.el7_6.x86_64)
15 * Toolchain:        gcc (GNU Compiler Collection)
16 * C Compiler:       Version 4.8.5 (at /usr/bin/gcc)
17 * C++ Compiler:     Version 4.8.5 (at /usr/bin/g++)
18
19 Build performance summary:
20 * Cores to use:      1
21 * Memory limit:     8000 MB
22 * ccache status:     Active (3.3.4)
```

执行编译:

```
1 nohup make all &
```

由于编译是在云端且编译的时间比较长, 这里使用 `nohup` 命令避免在终端退出后结束编译进程, `&` 命令后台运行。编译时要保证至少有 **7GB+** 的硬盘空间, 避免存储不够导致编译失败, 本人就是在 **10GB** 的 **VPS** 上折腾了很久, 总是提示空间不够。另外由于本人云主机的配置非常有限, 编译花费时间也是比较长的, 大概用了 **8** 个多小时才编译完毕。编译的时候一定要耐心, 有问题可以放一放, 过一天再来处理, 今天编译不通过明天继续。OpenJDK 8 编译不通过换 OpenJDK 10, Mac 编译不通过换 CentOS。

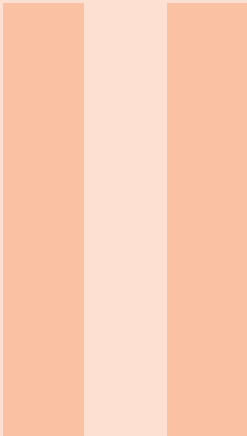
## 1.8 03 月

### 1.8.1 DbVisualizer

今天使用 PLSQL 连接 Oracle 9i 始终有问题，提示 error while trying to retrieve text for error ORA-12154。始终没有找到问题所在，最后用 DbVisualizer 连接上了 Oracle。截止到 2019 年 3 月，Oracle 9i 已经发布有 18 年<sup>110</sup>了，较老的数据库驱动都不好，新版发布的应用兼容性也没有把握。解决问题的过程中，特别吐槽 Oracle 官方网站的驱动下载体验，还有 CSDN 设计的狗屎积分下载，跟百度文库一个德行，贼恶心。

---

<sup>110</sup><https://dba.stackexchange.com/questions/232706/error-while-trying-to-retrieve-text-for-error-ora-12154>



# 菜单

<b>2</b>	<b>主食 .....</b>	<b>171</b>
2.1	饭	
2.2	菜	





## 2. 主食

### 2.1 饭

#### 2.1.1 红薯粥 (Sweet Potato Congee)

红薯粥简单方便，夏天正适宜，做的时间不需要限制 (可以做好放冰箱过几个小时再吃)，也不需要刻意操心 (关注火候或者时间)。需要改进的地方是粘稠度还较欠缺，下次可以尝试多多搅拌试试。食材方面，不知是红薯的缘故还是做法的缘故，自然的甜度还不足，不想放糖增加甜度，能够有自然的甜度最佳。

### 2.1.2 南瓜粥 (规划中)

### 2.1.3 手抓饼 (规划中)

## 2.2 菜

### 2.2.1 油豆腐粉丝汤

材料需要油豆腐、粉丝、酱油<sup>1</sup>、白砂糖、盐、色拉油、大葱。

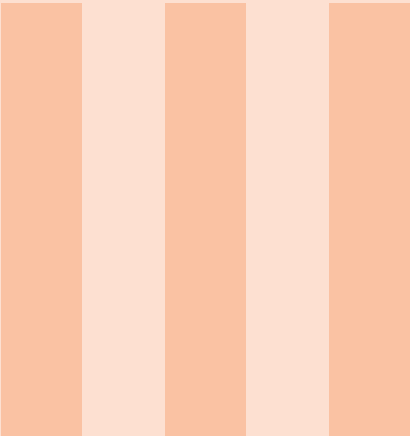
### 2.2.2 凉拌黄瓜

优点是简单方便，没有油，有油的菜洗碗较麻烦，尽量避免。调料不够专业 (蒜 + 蚝油 + 生抽 + 小米辣 + 酱油 + 小葱 + 香菜 + 醋 + 白糖 + 盐 + 老抽 + 味精 + 芝麻香油)，小米辣可以去掉，小葱香菜无法存放，暂不考虑。

---

<sup>1</sup>酱油俗称豉油，主要由大豆、淀粉、小麦、食盐经过制油、发酵等程序酿制而成的。以咸味为主，亦有鲜味、香味等。它能增加和改善菜肴的口味，还能增添或改变菜肴的色泽。我国人民在数千年前就已经掌握酿制工艺了。酱油一般有老抽和生抽两种：老抽较咸，用于提色；生抽用于提鲜。在烹饪绿色蔬菜时不必放酱油，因为酱油会使这些蔬菜的色泽变得黑褐暗淡，并失去了蔬菜原有的清香。另外，“烹调酱油”未经加热不宜直接食用。





# Book

<b>3</b>	<b>Paper .....</b>	<b>177</b>
3.1	Font	
3.2	纸的种类	
	<b>Bibliography .....</b>	<b>179</b>
	Books	
	Articles	



人类从古至今，总共有多少本书，答案是  $129,864,880^2$ 。

---

<sup>2</sup><http://booksearch.blogspot.com/2010/08/books-of-world-stand-up-and-be-counted.html>





## 3. Paper

### 3.1 Font

#### 3.1.1 类型

### 3.2 纸的种类

#### 3.2.1 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<sup>1</sup>。

---

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

### 3.2.2 连史纸

### 3.2.3 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**

