ThoughtWorks® | IT大咖说 知识分享平台

BQConf

BETTER QUALITY Conference 2017
中国软件质量大会

软件质量关注者的分享和交流平台

ThoughtWorks®

ThoughtWorks® | IT大咖说 知识分享平台

BQConf

中国软件质量大会

DOCKER引领软件测试革新

孙远 华为中央软件院

ThoughtWorks®

我的经历

美国风河系统公司

- 负责linux build system、analysis tools、workbench测试工作。

华为中央软件院

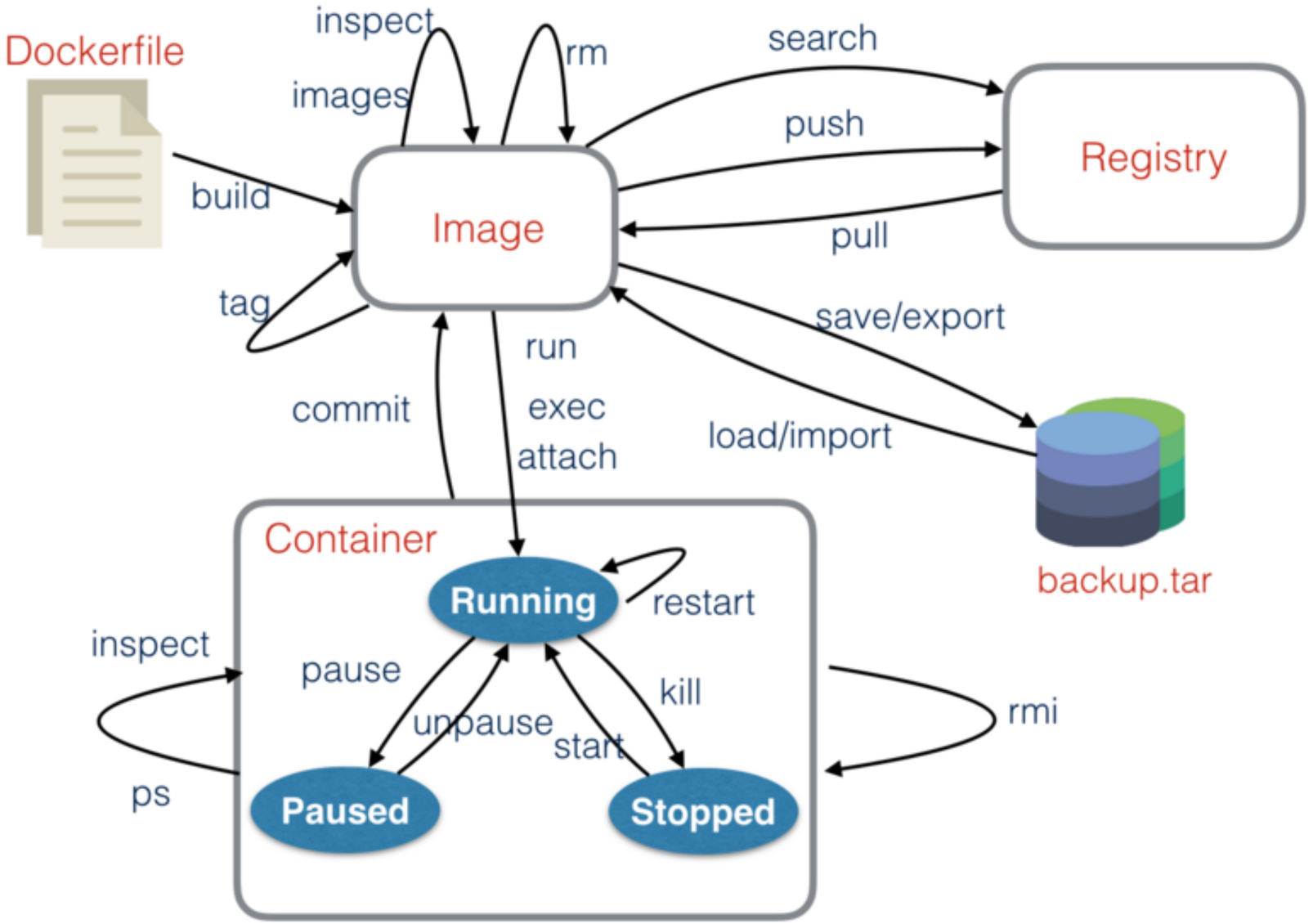
- 负责容器OS、docker测试工作，并参与项目过程改进。
- 参与开源社区：补充ltp社区user namespace特性测试用例和docker社区中的测试用例。
- 完成《docker进阶与实战》测试章节的编写工作。
- 支撑公司多个测试团队进行软件测试容器化改造。

内容

业务痛点

测试环境搭建时间长，难以保证环境的一致性。
测试执行时间长，难以满足快速迭代需求。
测试工具部署时间长，且成功率低。
源码编译速度慢，难以满足业务要求。
资源利用率低，物料浪费严重。
缺陷难以复现。
各项目组间的工程能力难以拉通。

DOCKER基础知识



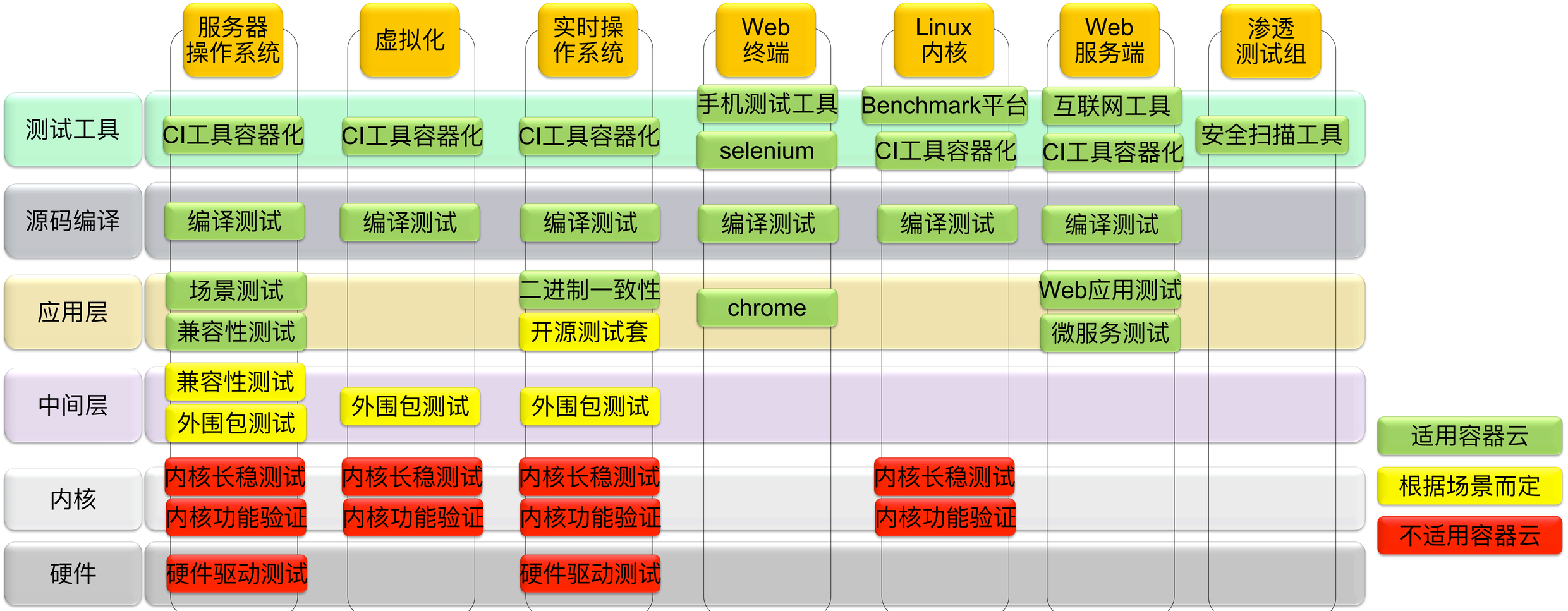
容器、镜像、仓库交互关系

DOCKER特点简述

	Docker	VM
•快速部署		
•资源限制	Yes	Yes
•环境隔离		
•弹性伸缩	Yes	No

	Docker	VM
•环境共享与迁移		
•高资源利用率		
•设备直通	Yes	Yes
•Dockerfile管理	Yes	No

工程能力容器化沙盘

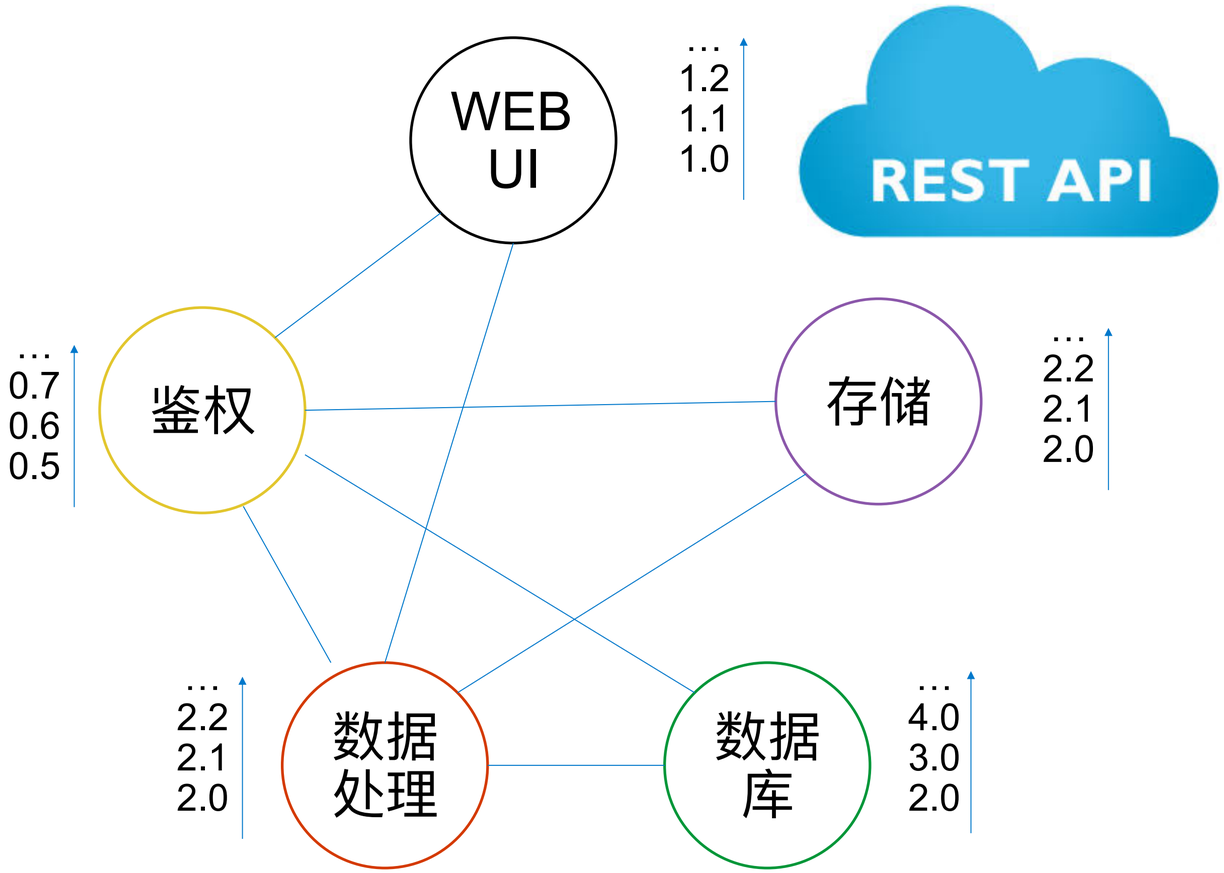


备注：容器云意味着其配置有一致的内核，相近的硬件资源。容器云和容器是两个概念。不适用容器云的业务并不代表不能使用容器进行业务改造。

容器化改造主要量化KPI

- 测试环境搭建时间、场景构建时间减少值
- 测试执行时间降低值
- 测试工具部署效率提升值
- 源码编译速度提升值
- 资源利用率提升值
- 节约物料成本
- 减少测试设计时间
- 软件缺陷复现概率

测试场景微服务化

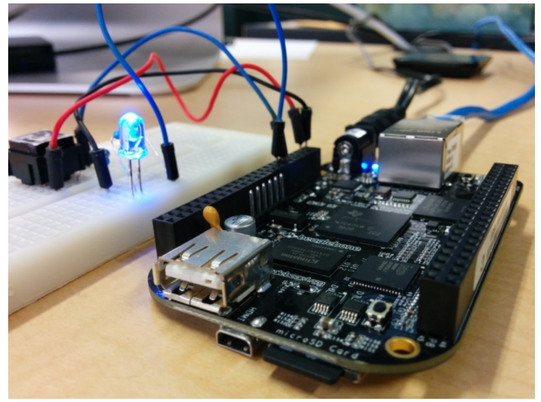

WEB UI
…
1.2
1.1
1.0
REST API
鉴权
…
0.7
0.6
0.5
存储
…
2.2
2.1
2.0
数据处理
…
2.2
2.1
2.0
数据库
…
4.0
3.0
2.0

测试工具容器化

提升测试工具可移植性，屏蔽不同Linux发行版。

测试工具部署

- 主机中（非被测机）
  可以部署一些CI工具、辅助测试工具（如发包工具）
  可选用较新的Linux发行版作为主机，通常支持Docker。



- 被测主机中
  可以适配需要运行在被测主机中的测试工具。（如fio压力测试工具）
  一些系统（特别是嵌入式系统）经过裁剪后文件系统较小，不一定默认包含Docker。需要在前期的需求中明确要包含Docker以保证可测试性。

软件测试执行加速方式

- 改进算法，优化测试用例。
- 选择功能强大主机，在主机中并行执行测试用例。
- 采用分布式的方式将测试任务下发到不同的主机中并行执行测试用例。

源码编译容器化

利用Docker快速部署、快速伸缩的特点，在短时间内获取容器云中大量系统资源支撑测试执行以达到测试加速的目的。在测试执行完之后，可快速释放所占资源，减少测试成本。

利用Dockerfile梳理编译环境配置步骤，使得环境准备的步骤更加清晰。

不必为每一个发行版的编译环境逐一配置虚拟机，只需配置好编译镜像即可，可将各业务组的编译资源集中使用，可以达到提高主机利用率和节约成本的目的。

Dockerfile:

```
FROM ubuntu:14.04
MAINTAINER Yuan Sun<sunyuan3@huawei.com>

# copy apt-get sources list to image
COPY sources.list /etc/apt/

# keep ubuntu up-to-date
RUN apt-get update && apt-get -qqy upgrade

# install tools for compile
RUN apt-get update && apt-get install -y \
    build-essential \
    automake \
    libtool \
    git
```

源码编译容器化

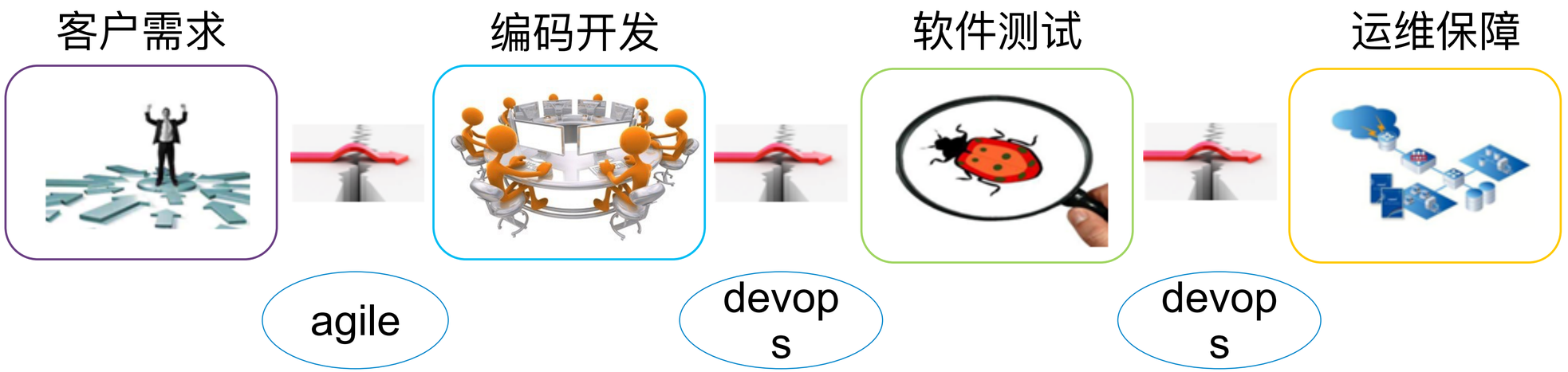
痛点：资源利用率不高
各项目组单独配置昂贵、性能好的主机进行源码编译。虽然编译速度有很大提升，但往往在没有转测试的时候，编译主机经常长时间空闲。很多时候空闲甚至达到90%以上，主机无法“吃饱”。

解决方案：
建立资源池，利用容器云实现源码编译任务按需分配。可以减少硬件设备采购成本50%以上。
开发转测试交付由单纯的“源码交付”改为“源码与编译环境镜像交付”。
使用docker取代chroot来进行资源管理。

DEVOPS

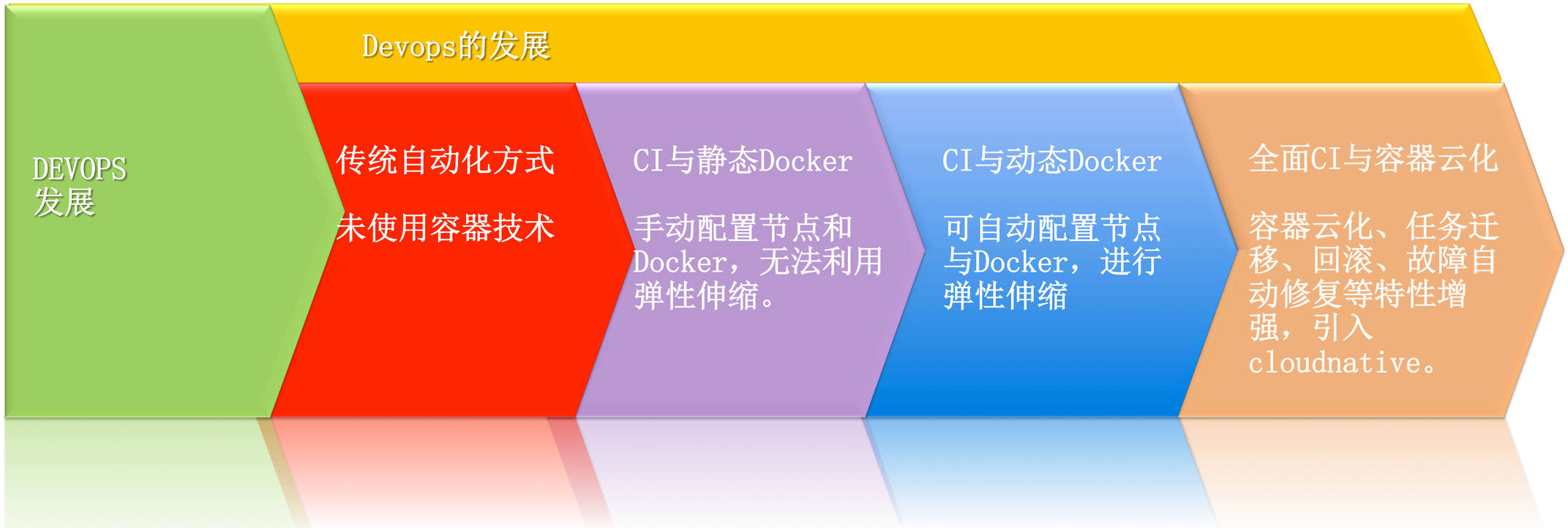
痛点

- 开发：无法复现测试提交的缺陷单、测试周期长、无法有效及时获取运维环境和数据。
- 测试：开发常常驳回问题单、开发需要不明不易理解、难以了解生产环境。
- 运维：开发提供的交付件不稳定、迭代速度快无法保障系统稳定性。



DEVOPS

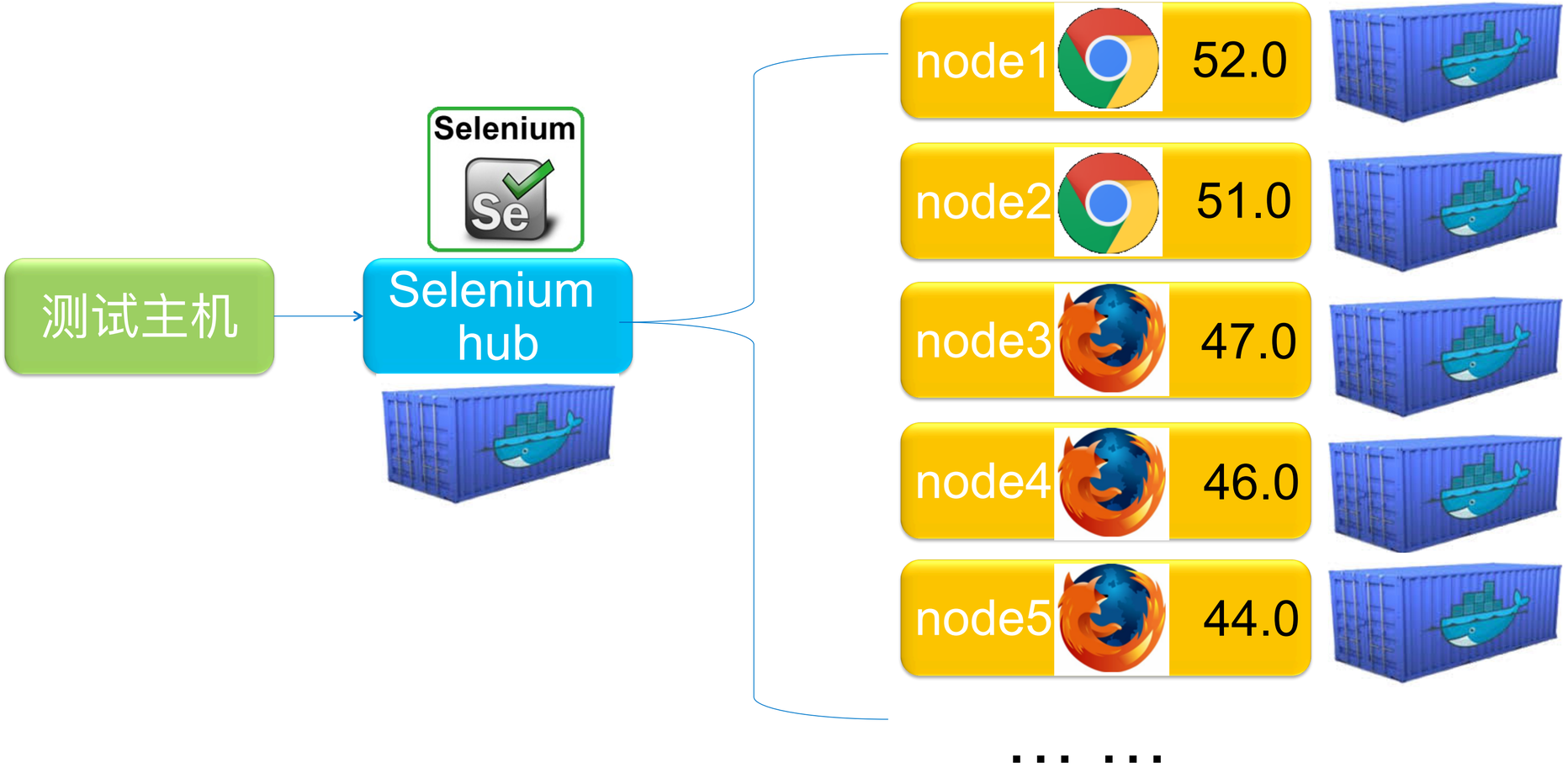
Devops发展的里程碑



应用层软件测试容器化

将Docker和Devops结合起来使用，统一开发、测试、运维环境，打通全流程自动化任务。

利用容器云中Docker可扩展可伸缩的特性加速测试执行或者模拟多个终端用户来进行测试。



应用层软件测试容器化

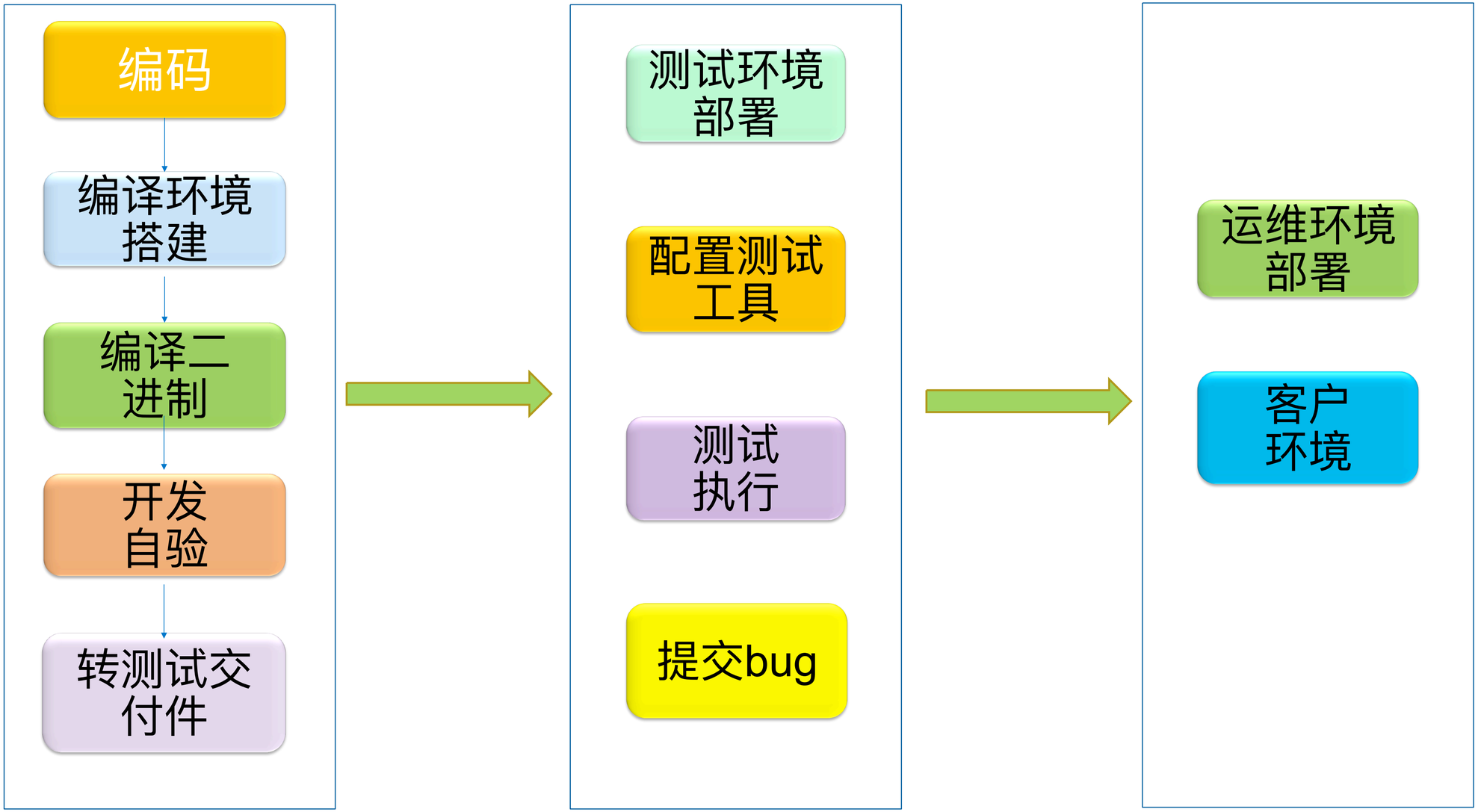
利用容器隔离性和快速环境清理的特性，并行执行软件兼容性测试(如安装不同版本的数据库)。

通过容器模拟测试执行需要的外部环境。可将一些典型的测试场景进行容器化处理，将镜像存储到镜像仓库中。

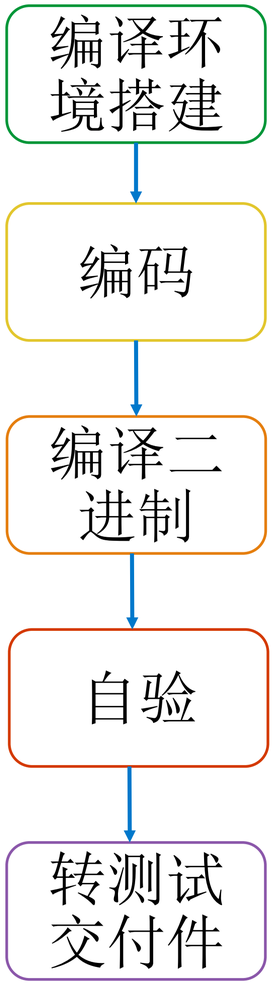
可以通过docker-compose以微服务的方式部署这些测试场景，便于其他团队复用已有的测试场景。

如想在容器云中运行与内核相关的应用，可将应用程序进行内核特性解耦，将内核模块对应的部分代码移植到应用程序中。

应用层软件测试容器化


编码
编译环境搭建
编译二进制
开发自验
转测试交付件
测试环境部署
配置测试工具
测试执行
提交bug
运维环境部署
客户环境

应用层软件测试容器化



1

手动搭建环境，难以保证环境一致。

编码环境不易配置，如vim中安装golang控件。

无法实现编译动态资源扩展，不易加速。

搭建自验环境较长，开发自验往往不充分。在同一套环境中往往对不同交付件进行验证，容易造成脏数据，污染环境。

交付件为源码或二进制，不仅环境不易不统一，而且部署复杂。

容器化前

2

可通过dockerfile梳理编译环境搭建步骤，确保环境一致。

秒级部署编码环境。

通过底层docker update命令可以实现资源动态扩展，加速效果可观。

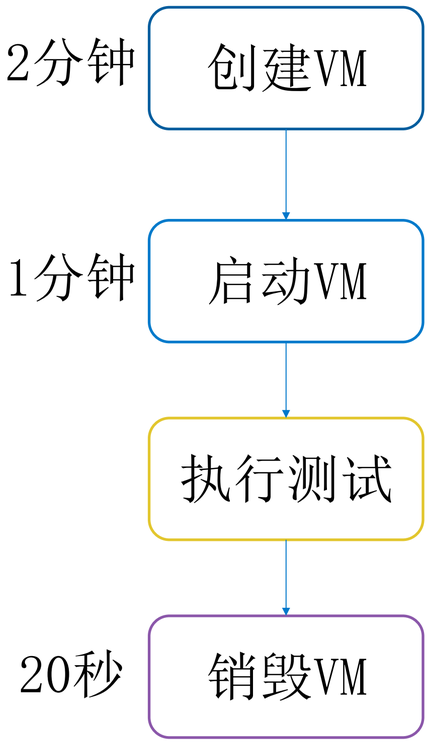
通过docker镜像打包测试环境，可以秒级配置环境。并通过web hook强制执行自动化测试。有力的支撑了测试迁移。Docker提供了清洁的环境。

交付件为docker镜像，可以秒级部署。

容器化后

应用层软件测试容器化

测试环境部署

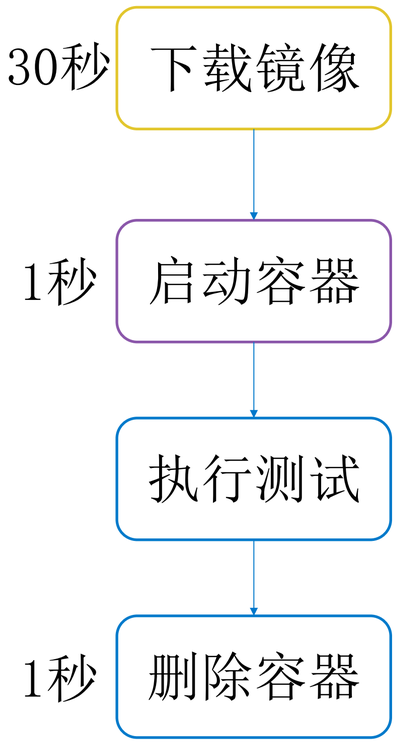


1

测试环境搭建费时费力，为了保证环境清洁性需要频繁创建VM。否则环境污染问题会累积。

配合多套环境时，时间将乘以N。

容器化前



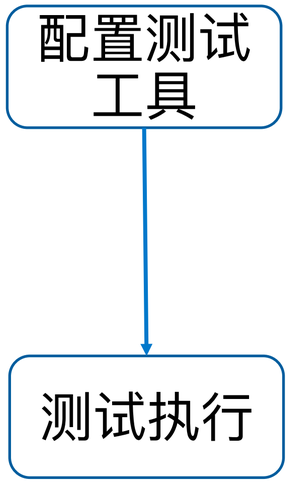
2

将测试环境打包为镜像，可以秒级部署。可以轻松创建多套环境。

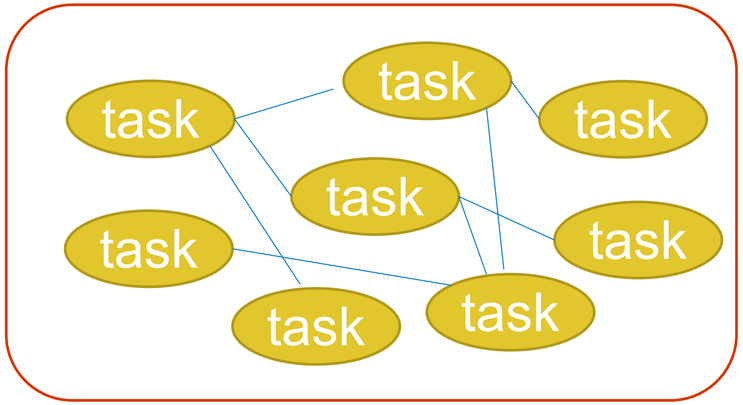
配合多套环境时，几乎不增加时间。

容器化后

应用层软件测试容器化



配置测试工具工作复杂，成功率低。

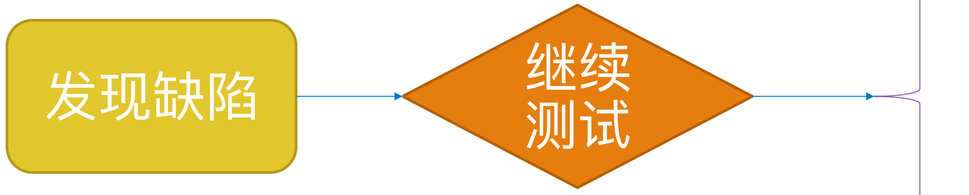


多任务共享主机资源，环境污染难以避免；测试资源难以快速扩展。

测试工具容器化后可以秒级部署，成功率高。

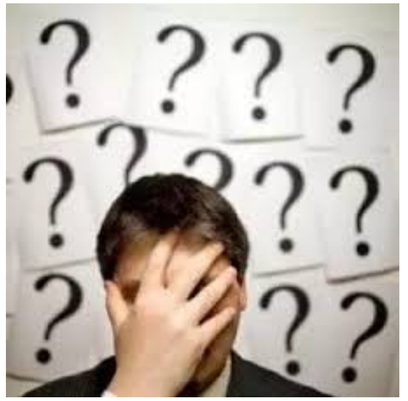
利用容器可以隔离测试环境，利用容器云可以非常容易的自行资源扩展和伸缩，最大限度的并行执行测试。

应用层软件测试容器化

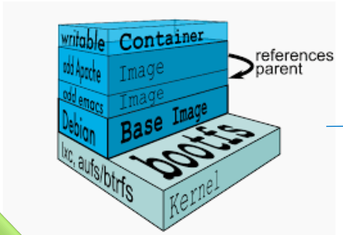


Y: 缺陷现场难以保留，其它测试程序可以会污染环境。

N: 其它测试任务不得不中断。



- 通过docker commit将环境打包成为镜像上传到仓库中并提供给开发团队。
- 启动新的容器执行其它测试任务。



docker commit

docker push

应用层软件测试容器化

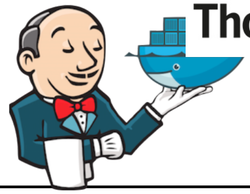
运维环境部署

- 容器化前：运维难以解决业务峰值。环境出问题后，开发团队不易复现。
- 容器化后：可以通过cloud native解决业务峰值。开发团队可以很容易的复现缺陷。

客户环境

- 容器化前：客户环境出现问题，开发团队难以复现。
- 容器化后：可以通过docker save将环境打包成为镜像。

DOCKER与JENKINS



ThoughtWorks | IT大咖说 知识分享平台

Docker插件	功能
Docker Commons Plugin	为其它docker插件提供api
Docker Plugin	可以使用Docker主机动态分配的容器作为Jenkins的从节点。
Docker Slaves Plugin	使用容器来配置构建agents，对于镜像的使用没限制。
Docker Pipeline Plugin	允许构建和使用pipelines中的容器。
Kubernetes Plugin	通过由Kubernetes管理的多个Docker主机系统来动态分配的容器作为Jenkins的从节点。
Mesos Plugin	使Jenkins可以根据工作负载动态启动 Mesos集群中的Jenkins slaves。
Swarm Plugin	使slaves可以自动发现Jenkins master并自动加入。

移动终端测试容器化

移动终端测试容器化

痛点分析：

工程师通常只对自己的测试环境熟悉。

测试环境和设备无法共用，无法有效发挥自动化价值。

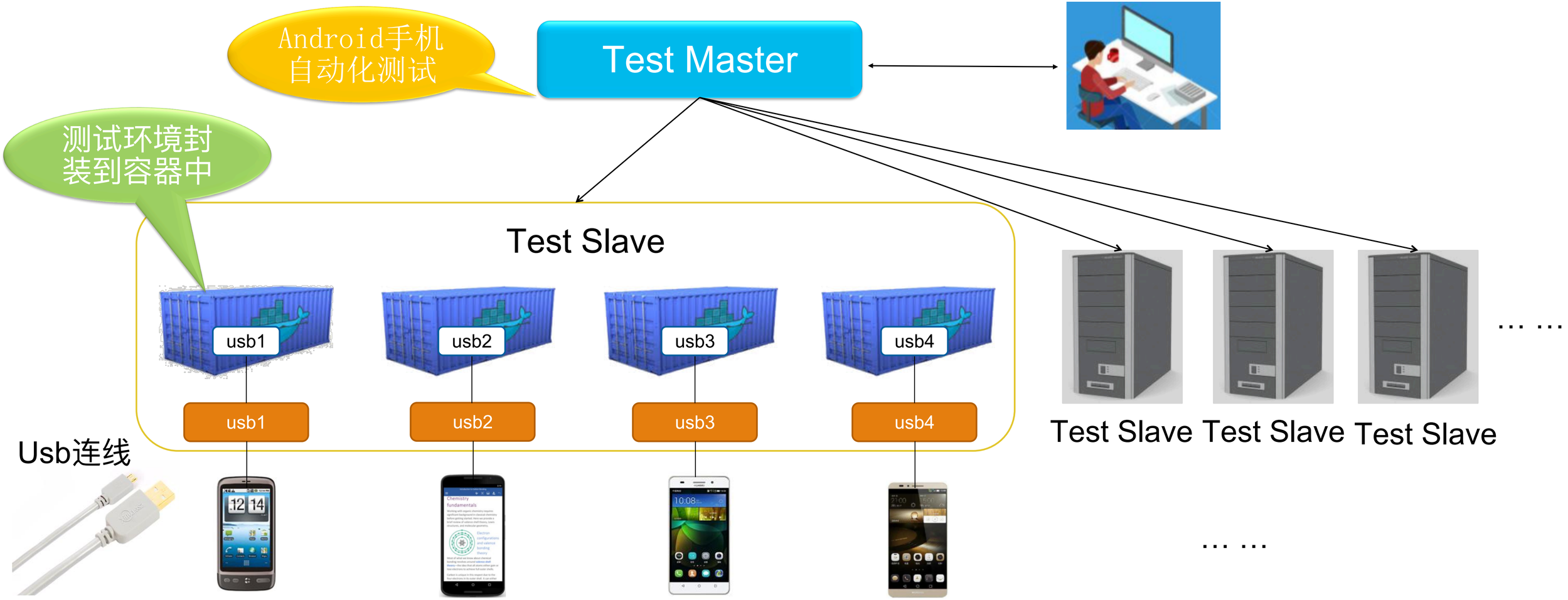
测试环境与被测终端测试任务无法避免污染。

移动终端测试容器化

技术挑战：

需要充分发挥自动化测试价值。
测试环境需要能够复用。
有效避免环境污染，测试任务不受到其它因素的干扰。
充分利用硬件资源。
任何人在任何时间任何地点可以在任何设备中运行任何自动化用例。
满足开发自验、CI、缺陷复现等需求。

移动终端测试容器化




```
docker run -ti --device=/dev/bus/usb/002/001 ubuntu:test /bin/bash
```

移动终端测试容器化

难点：测试时手机重启导致硬件设备号更改，而docker不支持动态加载硬件设备。

移动终端测试容器化

解决方法：

1. 求助社区，添加新的特性。
2. 停止当前测试容器，启动新的容器来加载新的设备号。
3. 修改/etc/udev/rules.d/70-persistent-net.rules使手机重启后设备号不变。
4. 通过底层device cgroup特性动态加载新的设备号。

移动终端测试容器化

动态加载实现原理：

启动容器并获取设备信息，开始测试。

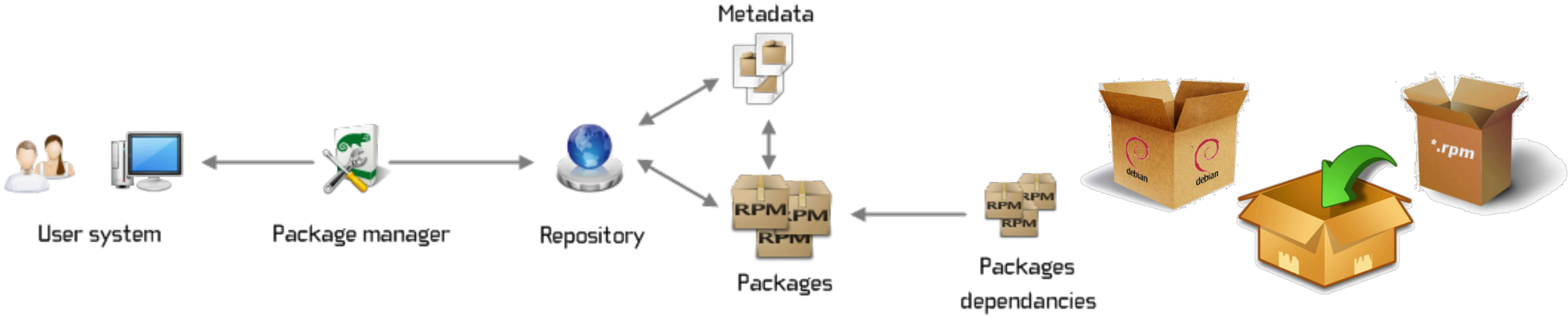
手机重启后，遍历所有usb 设备，定位新的设备号。

将设备加入到device cgroup允许容器访问的列表中，并在容器中创建设备节点。

中间层软件测试容器化

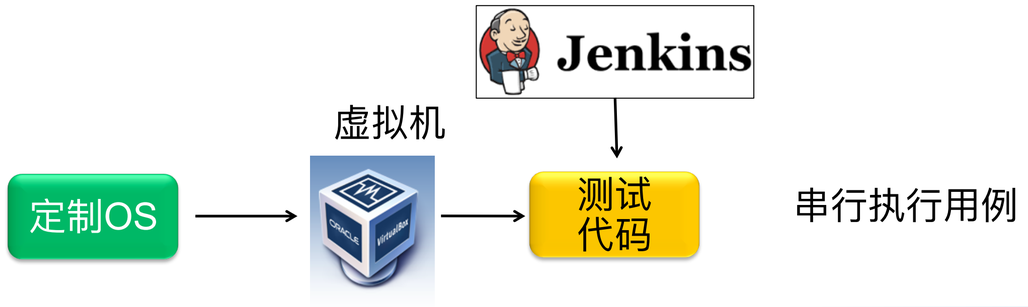
大部分Linux外围包（通常是rpm或deb包）与内核无强依赖，是适合使用容器云进行加速的。

因为在容器云中无法自定义内核，与内核相关性强的外围包的测试是不适用容器云。解决方法：搭建可定制内核的私有云

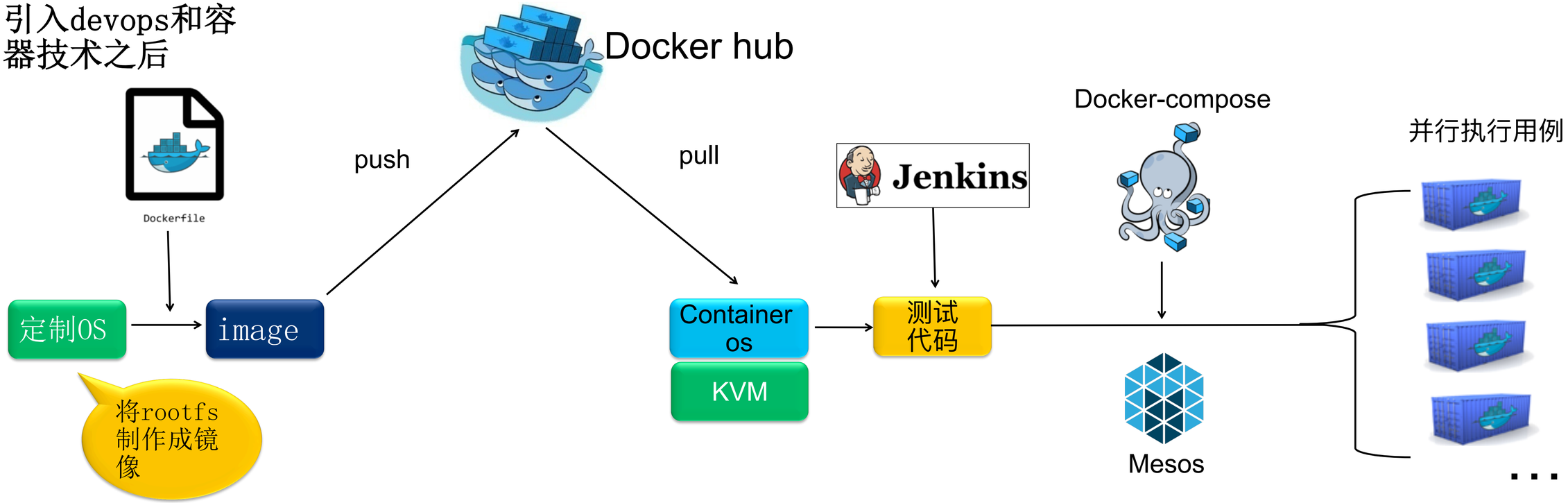


中间层软件测试容器化

引入devops和容器技术之前



引入devops和容器技术之后



中间层软件测试容器化

特例：

(1)grub包:grub是多系统启动规范的实现，因为与主机启动强相关。

(2)ntp包:ntp是网络时间协议(Network Time Protocol)，用来同步网络中各个计算机的时间的协议。

当前内核不支持time namespace，即容器与主机间、容器与容器间不能使用不同的时间（注意是时间不是时区）。

(3)与init有强相关的包。

中间层软件测试容器化

yum安装与兼容性测试

业务痛点

- 测试用例多，无法在清洁的环境中一一覆盖。
- 构建测试场景花费时间长。
- 用例间依赖关系复杂。

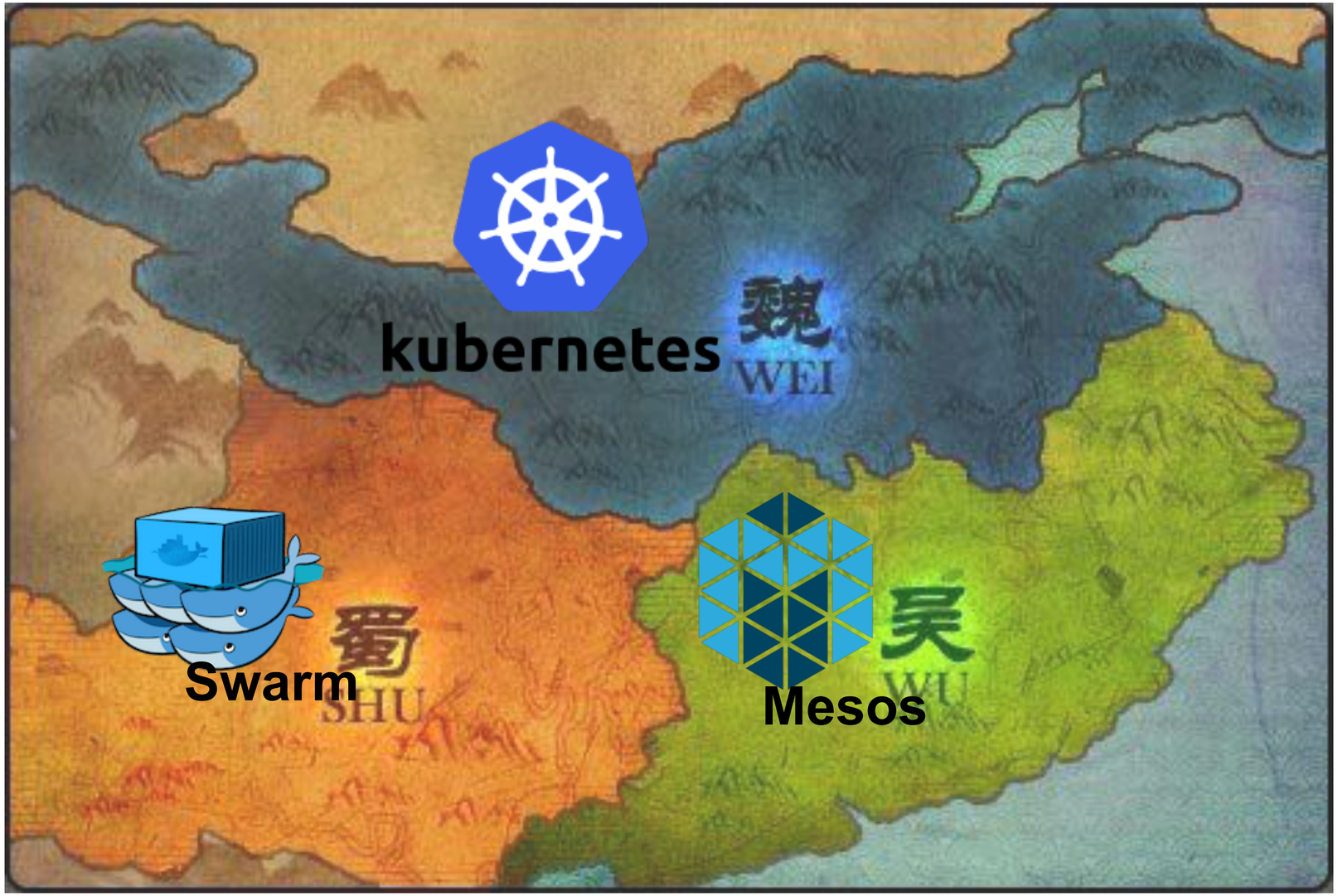
遇到的问题

- 批量启动容器后，容器启动速度变慢。
- 磁盘IO成为制约用例速度的瓶颈。

解决方法

- 利用overlay取代devicemapper作为Storage Driver。
- 将tmpfs挂载到/var/lib/docker目录中。

KUBERNETES SWARM MESOS



内核测试容器化

内核测试容器化改造方案：

- Ltp测试套中的用例默认是串行执行的，可启动多个容器并发执行内核功能测试来提高cpu负载，以便减少测试执行时间。

- 可以在容器内搭建内核测试环境。

Ltp测试套中的测试用例源码默认是被动态编译的，其运行需要一些动态链接库。而很多系统特别是嵌入式系统往往缺少这些动态链接库，导致部分内核测试用例无法执行。可将必要的动态库集成到Docker镜像中，这样既保证了测试用例可以顺利执行，又可以避免在主机中直接安装库文件导致环境污染。

- 使用Docker构建内核集成测试场景。

因为内核是底层技术，很难找到一个能够包含多个内核特性的集成测试场景。通过以下Docker命令可以轻松搭建namespace、cgroup、capability、seccomp等内核特性的集成测试场景，大大减少集成测试设计时间。

```
$ docker run --memory 20m --cap-add sys_admin --security-opt seccomp=unconfined --userns-remap default ubuntu:14.04 bash
```

内核测试容器化

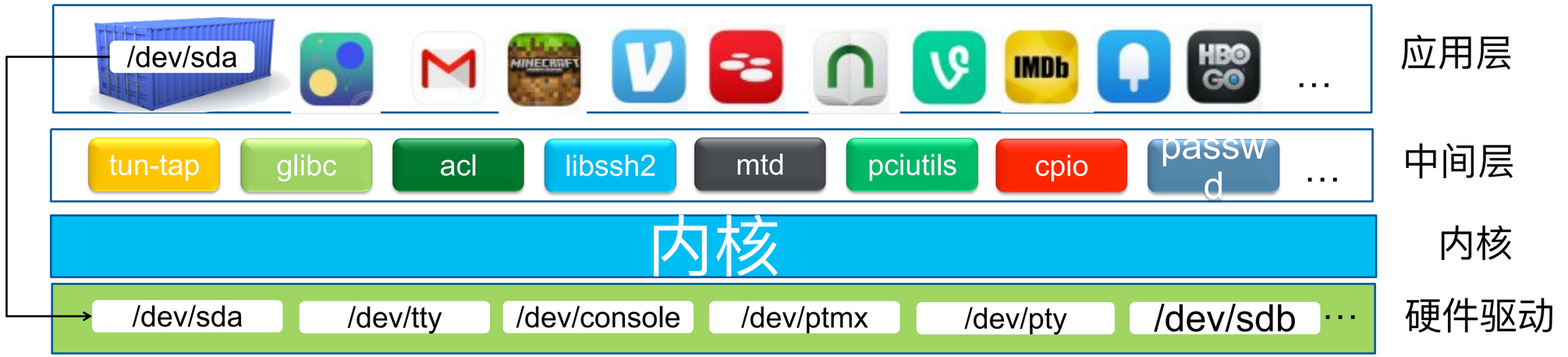
局限性：

1. 在容器内执行内核测试时会有部分用例执行失败。例如，pidns32测试用例是用来测试namespace的最大嵌套深度，但容器已经嵌套了一层namespace。
2. 某些内核测试项是排他的，需要单独运行，这部分测试需要进行额外的操作处理，比如一些中断操作、寄存器操作。
3. 内核性能测试的执行不适用容器化，否则会有失真。

Ltp(linux test project)测试套pidns32测试用例

```
#define MAXNEST 32
...
static int child_fn1(void *arg)
{
...
        if (level == MAXNEST)
                return 0;
        cpid1 = ltp_clone_quick(CLONE_NEWPID | SIGCHLD,
                (void *)child_fn1, (void *)(level + 1));
}
```

硬件驱动测试容器化



- 难点：Docker的设计初衷是来屏蔽各硬件平台差异的。
- 方案：Docker利用内核的device cgroup特性可以实现设备直通的功能，即可将主机中的设备映射到容器中进行测试。
- 收益：测试环境不会污染主机环境；便于设备驱动并行测试；即使主机文件系统缺少测试所需的库文件，仍然可以使用容器的文件系统中的工具来辅助测试。

例子：磁盘块设备测试

```
$ docker run -device /dev/sda:/dev/sda ${your_image} ${your_test_script}
```

压力稳定性测试痛点

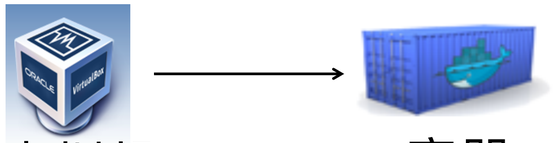
•部署工作复杂。
加压测试套编译与运行有很强的平台依赖性，当测试涉及多平台复杂组网时，会出现很多兼容性问题。系统巡检工具的安装依赖包多，需要人工进行各个版本依赖包的适配工作。一套完整的长稳测试框架的部署（包括背景加压、系统巡检与用例执行）会耗费大量人力，影响了测试效率。

•加压模型单一，多个模型不易叠加。

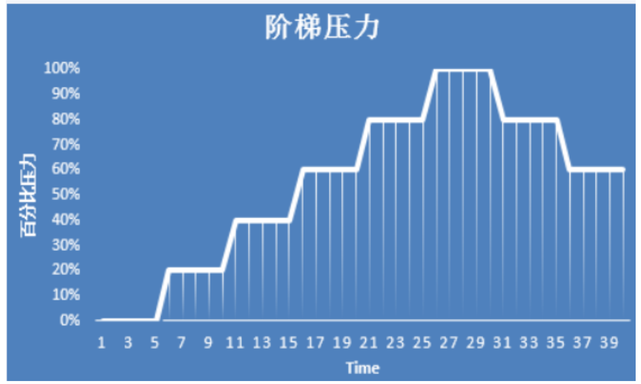
•容错能力差。
加压程序有可能本身存在内存泄露，导致开发团队不认可测试结果。加压程序垮掉时往往需要借助外部环境恢复。

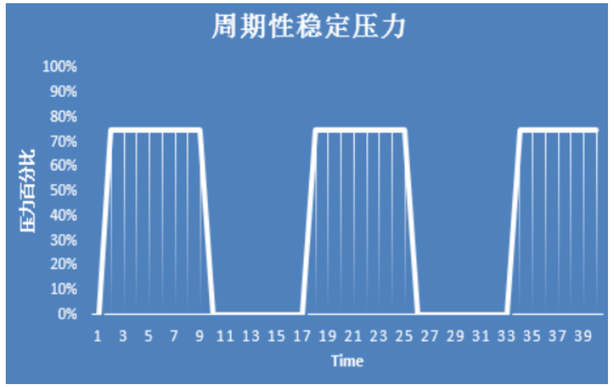
压力稳定性测试容器化

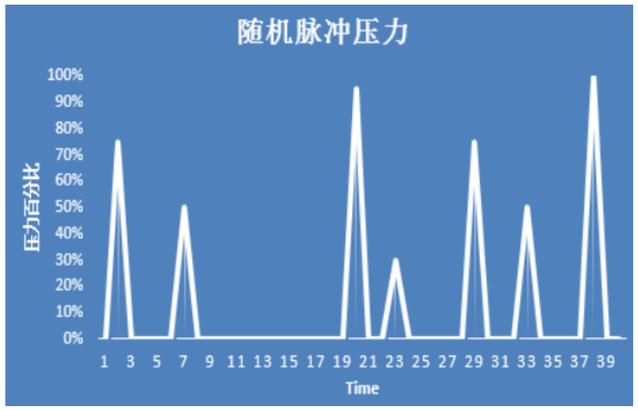
压力稳定性测试容器化
●CPU负载
-按照加压模型，对cpu占有率按照30%~80%
●内存负载
-按照加压模型，对内存占有率按照50%~80%进行规划
●IO频率
-使用fio工具

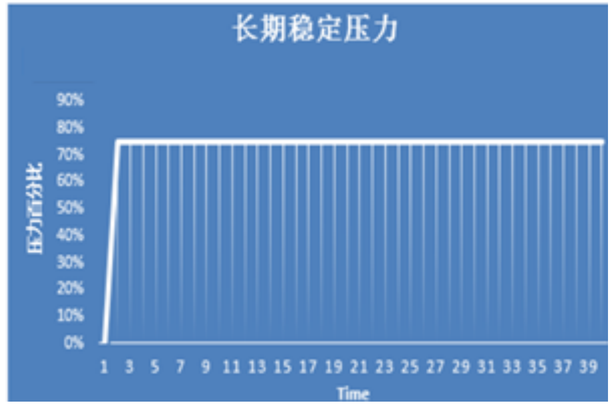


确保测试运行的更加平稳、持续；将测试工具内存泄露等问题带来的负面影响降到最低。









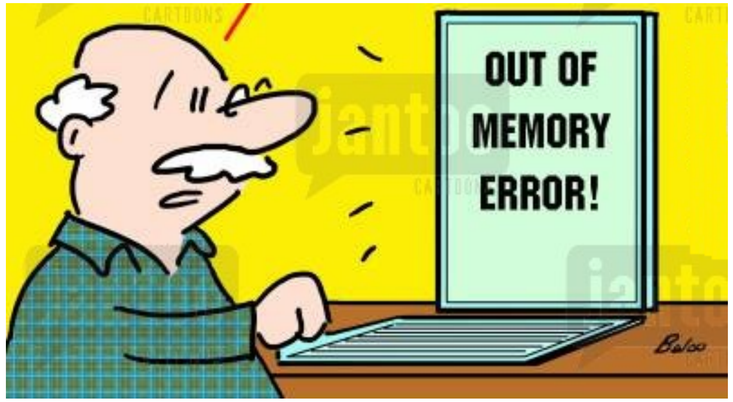
压力稳定性测试容器化

利用容器中的文件系统支撑加压程序运行，可以使长稳测试运行更加标准化，便于拉通多个测试组的测试能力。

容器化可以避免因加压程序导致系统垮掉的问题。

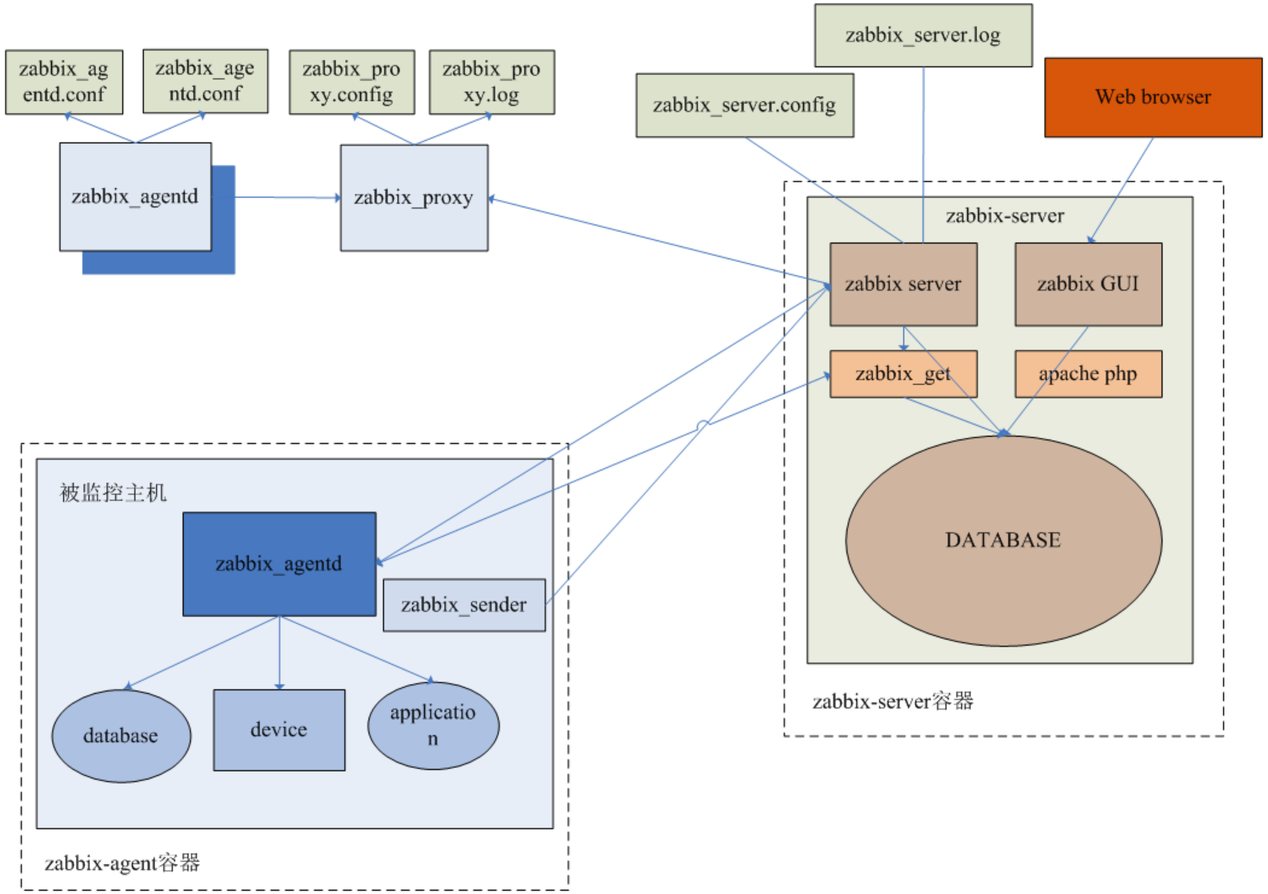
加压程序有内存泄露时，当容器试图使用超过限定内存时会触发OOM，只要设定好restart规则后容器所占资源会被清理同时容器会快速重新启动并持续进行加压。

```
$ docker run -tid -m 3g --restart=always \
 back_stress bash \
-c  “cpu_stress/test.sh”
```



压力稳定性测试容器化

系统巡检的容器化。将zabbix巡检工具的server端与agent端均打包为镜像，在多平台组网复杂的环境中，可以完全屏蔽平台化差异。



应用软件安装测试限制

问题

- 在容器中可以安装成功，为什么在主机中无法安装成功？

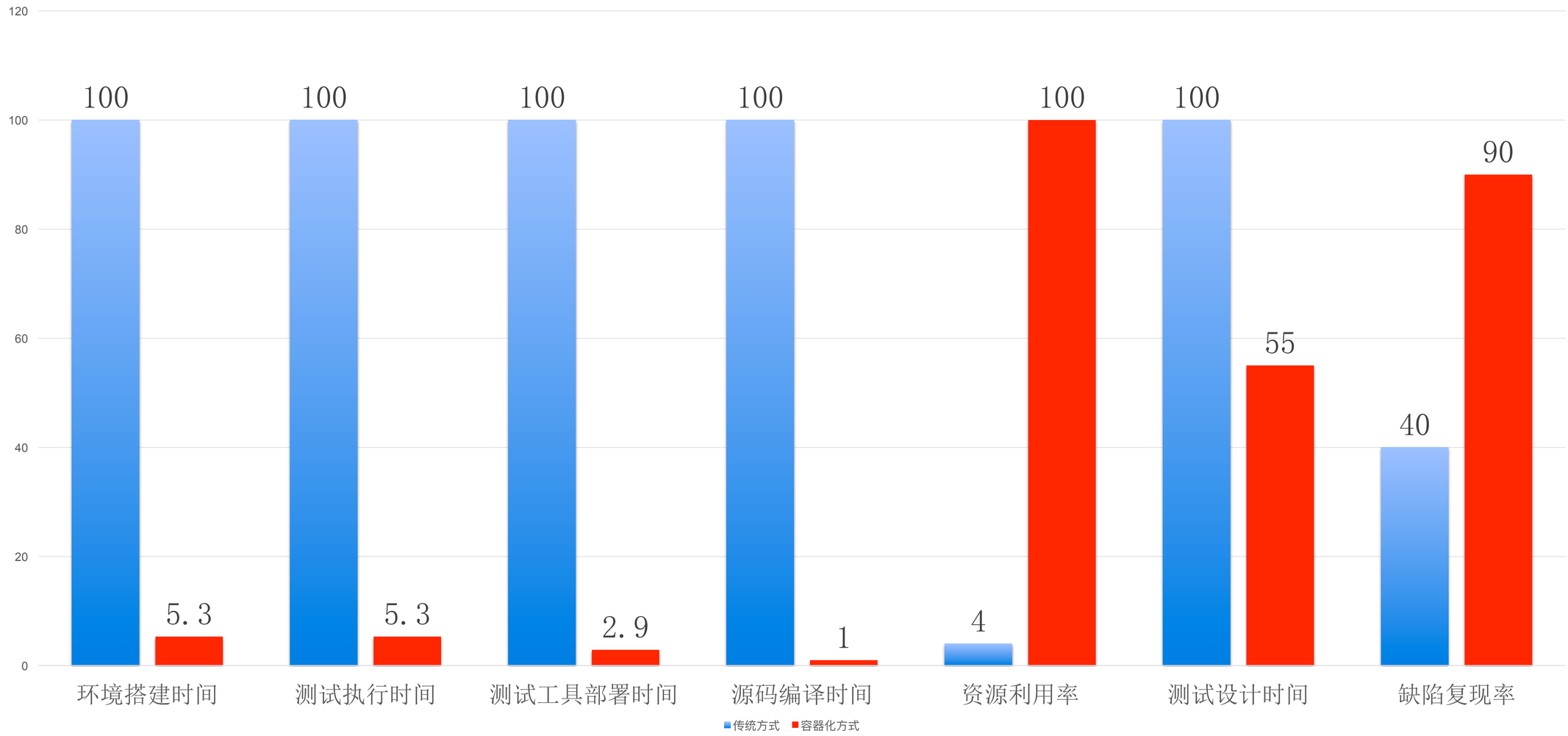
常见原因

- 容器镜像的默认安装包比虚拟机小很多，二者的文件系统是不同的。
- 容器和主机对应的内核版本以及其它未被隔离的信息不同。

解决方法

- 基于虚拟机的文件系统导出镜像。
- 编程时将程序与内核版本和其它未隔离的信息解耦。

改造前后对比图


120
100
80
60
40
20
0
100
100
100
100
100
100
5.3
5.3
2.9
1
4
55
40
90
环境搭建时间
测试执行时间
测试工具部署时间
源码编译时间
资源利用率
测试设计时间
缺陷复现率
传统方式
容器化方式

DOCKER实践KPI矩阵

	测试场景微服务化	测试工具容器化	测试执行加速	源码编译	应用层软件测试容器化	移动终端测试容器化	中间层软件测试容器化	内核测试容器化	硬件驱动测试容器化	压力稳定性测试容器化
环境搭建时间减少	95%＋			99%＋	99%+	90%+	95%+	90%+	90%+	99%+
测试执行时间降低					99%+		95%+		80%+	
测试工具部署时间减少		99%+			99%+	90%+	99%+		99%+	99%+
源码编译时间降低				99%＋	99%+					
资源利用率提升	95%＋	95%+	99%+	99%＋	99%+		95%+		90%+	
节约物料		20VM+		20VM+	40VM+	50VM+	50VM+			
减少测试设计时间					20%+			70%+		
缺陷复现率提升				50%+	50%+		50%+			

DOCKER实践矩阵

	测试场景微服务化	测试工具容器化	测试执行加速	源码编译	应用层软件测试容器化	移动终端测试容器化	中间层软件测试容器化	内核测试容器化	硬件驱动测试容器化	压力稳定性测试容器化
快速部署	✓	✓	✓	✓	✓		✓			✓
资源限制	✓	✓	✓	✓	✓		✓			✓
环境隔离	✓	✓	✓	✓	✓	✓	✓		✓	✓
弹性伸缩			✓	✓	✓		✓			
环境共享与迁移	✓	✓	✓	✓	✓	✓	✓			✓
高资源利用率	✓	✓	✓	✓	✓	✓	✓			
设备直通						✓			✓	
Dockerfile管理	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

ThoughtWorks® | IT大咖说 知识分享平台

BQConf

ThoughtWorks®