简介

如今，软件通常会作为一种服务来交付，它们被称为网络应用程序，或软件即服务（SaaS）。12-Factor 为构建如下的 SaaS 应用提供了方法论：

- 使用**标准化**流程自动配置，从而使新的开发者花费最少的学习成本加入这个项目。
- 和操作系统之间尽可能的**划清界限**，在各个系统中提供**最大的可移植性**。
- 适合**部署**在现代的**云计算平台**，从而在服务器和系统管理方面节省资源。
- 将开发环境和生产环境的**差异降至最低**，并使用**持续交付**实施敏捷开发。
- 可以在工具、架构和开发流程不发生明显变化的前提下实现**扩展**。

这套理论适用于任意语言和后端服务（数据库、消息队列、缓存等）开发的应用程序。

背景

本文的贡献者者参与过数以百计的应用程序的开发和部署，并通过 Heroku 平台间接见证了数十万应用程序的开发，运作以及扩展的过程。

本文综合了我们关于 SaaS 应用几乎所有的经验和智慧，是开发此类应用的理想实践标准，并特别关注于应用程序如何保持良性成长，开发者之间如何进行有效的代码协作，以及如何 避免软件污染 。

我们的初衷是分享在现代软件开发过程中发现的一些系统性问题，并加深对这些问题的认识。我们提供了讨论这些问题时所需的共享词汇，同时使用相关术语给出一套针对这些问题的广义解决方案。本文格式的灵感来自于 Martin Fowler 的书籍：*Patterns of Enterprise Application Architecture* ， *Refactoring* 。

读者应该是哪些人？

任何 SaaS 应用的开发人员。部署和管理此类应用的运维工程师。

12-factors

I. 基准代码

一份基准代码，多份部署

II. 依赖

显式声明依赖关系

III. 配置

在环境中存储配置

IV. 后端服务

把后端服务当作附加资源

V. 构建，发布，运行

严格分离构建和运行

VI. 进程

以一个或多个无状态进程运行应用

VII. 端口绑定

通过端口绑定提供服务

VIII. 并发

通过进程模型进行扩展

IX. 易处理

快速启动和优雅终止可最大化健壮性

X. 开发环境与线上环境等价

尽可能的保持开发，预发布，线上环境相同

XI. 日志

把日志当作事件流

XII. 管理进程

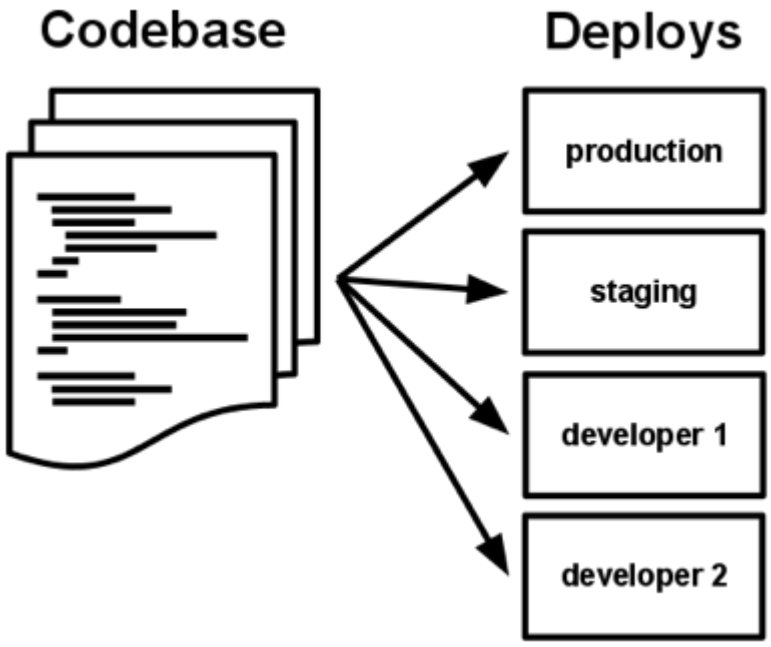
后台管理任务当作一次性进程运行

I. 基准代码

一份基准代码（*Codebase*），多份部署（*deploy*）

12-Factor 应用(译者注：应该是说一个使用本文概念来设计的应用，下同)通常会使用版本控制系统加以管理，如 Git,Mercurial, Subversion。一份用来跟踪代码所有修订版本的数据库被称作 *代码库*（code repository, code repo, repo）。

在类似 SVN 这样的集中式版本控制系统中，*基准代码* 就是指控制系统中的这一份代码库；而在 Git 那样的分布式版本控制系统中，*基准代码* 则是指最上游的那份代码库。



基准代码和应用之间总是保持一一对应的关系：

- 一旦有多个基准代码，就不能称为一个应用，而是一个分布式系统。分布式系统中的每一个组件都是一个应用，每一个应用可以分别使用 12-Factor 进行开发。
- 多个应用共享一份基准代码是有悖于 12-Factor 原则的。解决方案是将共享的代码拆分为独立的类库，然后使用 依赖管理 策略去加载它们。

尽管每个应用只对应一份基准代码，但可以同时存在多份部署。每份 *部署* 相当于运行了一个应用的实例。通常会有一个生产环境，一个或多个预发布环境。此

外，每个开发人员都会在自己本地环境运行一个应用实例，这些都相当于一份部署。

所有部署的基准代码相同，但每份部署可以使用其不同的版本。比如，开发人员可能有一些提交还没有同步至预发布环境；预发布环境也有一些提交没有同步至生产环境。但它们都共享一份基准代码，我们就认为它们只是相同应用的不同部署而已。

II. 依赖

显式声明依赖关系（*dependency*）

大多数编程语言都会提供一个打包系统，用来为各个类库提供打包服务，就像 Perl 的 CPAN 或是 Ruby 的 Rubygems。通过打包系统安装的类库可以是系统级的（称之为“site packages”），或仅供某个应用程序使用，部署在相应的目录中（称之为“vendoring”或“bunding”）。

12-Factor 规则下的应用程序不会隐式依赖系统级的类库。它一定通过 *依赖清单*，确切地声明所有依赖项。此外，在运行过程中通过 *依赖隔离* 工具来确保程序不会调用系统中存在但清单中未声明的依赖项。这一做法会统一应用到生产和开发环境。

例如，Ruby 的 Gem Bundler 使用 `Gemfile` 作为依赖项声明清单，使用 `bundle exec` 来进行依赖隔离。Python 中则可分别使用两种工具 – Pip 用作依赖声明，Virtualenv 用作依赖隔离。甚至 C 语言也有类似工具，Autoconf 用作依赖声明，静态链接库用作依赖隔离。无论用什么工具，依赖声明和依赖隔离必须一起使用，否则无法满足 12-Factor 规范。

显式声明依赖的优点之一是为新进开发者简化了环境配置流程。新进开发者可以检出应用程序的基准代码，安装编程语言环境和它对应的依赖管理工具，只需通过一个 *构建命令* 来安装所有的依赖项，即可开始工作。例如，Ruby/Bundler 下使用 `bundle install`，而 Clojure/Leiningen 则是 `lein deps`。

12-Factor 应用同样不会隐式依赖某些系统工具，如 ImageMagick 或是 `curl`。即使这些工具存在于几乎所有系统，但终究无法保证所有未来的系统都能支持应用顺利运行，或是能够和应用兼容。如果应用必须使用到某些系统工具，那么这些工具应该被包含在应用之中。

III. 配置

在环境中存储配置

通常，应用的 *配置* 在不同 部署 (预发布、生产环境、开发环境等等)间会有很大差异。这其中包括：

- 数据库，Memcached，以及其他 后端服务 的配置
- 第三方服务的证书，如 Amazon S3、Twitter 等
- 每份部署特有的配置，如域名等

有些应用在代码中使用常量保存配置，这与 12-Factor 所要求的**代码和配置严格分离**显然大相径庭。配置文件在各部署间存在大幅差异，代码却完全一致。

判断一个应用是否正确地将配置排除在代码之外，一个简单的方法是看该应用的基准代码是否可以立刻开源，而不用担心会暴露任何敏感的信息。

需要指出的是，这里定义的“配置”并**不**包括应用的内部配置，比如 Rails 的 `config/routes.rb`，或是使用 Spring 时 代码模块间的依赖注入关系 。这类配置在不同部署间不存在差异，所以应该写入代码。

另外一个解决方法是使用配置文件，但不把它们纳入版本控制系统，就像 Rails 的 `config/database.yml` 。这相对于在代码中使用常量已经是长足进步，但仍然有缺点：总是会不小心将配置文件签入了代码库；配置文件的可能会分散在不同的目录，并有着不同的格式，这让找出一个地方来统一管理所有配置变的不太现实。更糟的是，这些格式通常是语言或框架特定的。

12-Factor 推荐将应用的配置存储于 *环境变量* 中（ *env vars*, *env* ）。环境变量可以非常方便地在不同的部署间做修改，却不动一行代码；与配置文件不同，不小心把它们签入代码库的概率微乎其微；与一些传统的解决配置问题的机制（比如 Java 的属性配置文件）相比，环境变量与语言和系统无关。

配置管理的另一个方面是分组。有时应用会将配置按照特定部署进行分组（或叫做“环境”），例如 Rails 中的 `development`,`test`, 和 `production` 环境。这种方法无法轻易扩展：更多部署意味着更多新的环境，例如 `staging` 或 `qa`。 随着项

目的不断深入，开发人员可能还会添加他们自己的环境，比如 `joes-staging`，这将导致各种配置组合的激增，从而给管理部署增加了很多不确定因素。

12-Factor 应用中，环境变量的粒度要足够小，且相对独立。它们永远也不会组合成一个所谓的“环境”，而是独立存在于每个部署之中。当应用程序不断扩展，需要更多种类的部署时，这种配置管理方式能够做到平滑过渡。

IV. 后端服务

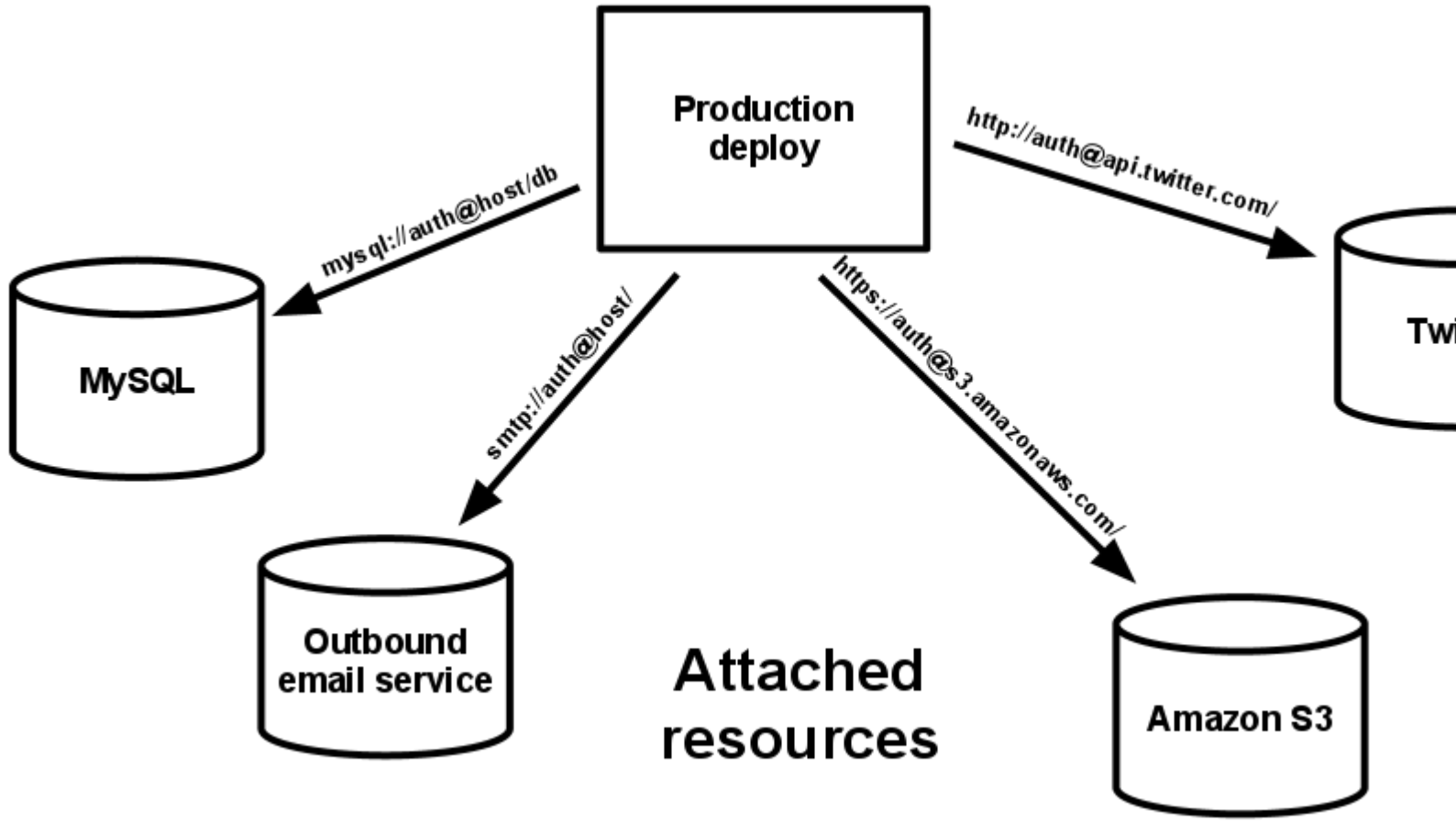
把后端服务(*backing services*)当作附加资源

*后端服务*是指程序运行所需要的通过网络调用的各种服务，如数据库（MySQL，CouchDB），消息/队列系统（RabbitMQ，Beanstalkd），SMTP 邮件发送服务（Postfix），以及缓存系统（Memcached）。

类似数据库的后端服务，通常由部署应用程序的系统管理员一起管理。除了本地服务之外，应用程序有可能使用了第三方发布和管理的服务。示例包括 SMTP（例如 Postmark），数据收集服务（例如 New Relic 或 Loggly），数据存储服务（如 Amazon S3），以及使用 API 访问的服务（例如 Twitter, Google Maps, Last.fm）。

12-Factor 应用不会区别对待本地或第三方服务。 对应用程序而言，两种都是附加资源，通过一个 url 或是其他存储在 配置 中的服务定位/服务证书来获取数据。12-Factor 应用的任意 部署，都应该可以在不进行任何代码改动的情况下，将本地 MySQL 数据库换成第三方服务（例如 Amazon RDS）。类似的，本地 SMTP 服务应该也可以和第三方 SMTP 服务（例如 Postmark ）互换。上述 2 个例子中，仅需修改配置中的资源地址。

每个不同的后端服务是一份 *资源*。例如，一个 MySQL 数据库是一个资源，两个 MySQL 数据库（用来数据分区）就被当作是 2 个不同的资源。12-Factor 应用将这些数据库都视作 *附加资源*，这些资源和它们附属的部署保持松耦合。



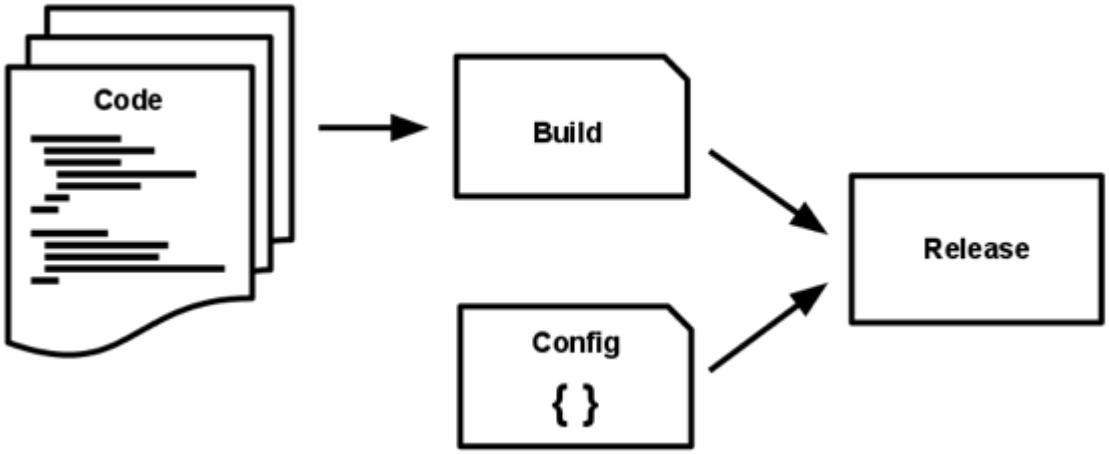
部署可以按需加载或卸载资源。例如，如果应用的数据库服务由于硬件问题出现异常，管理员可以从最近的备份中恢复一个数据库，卸载当前的数据库，然后加载新的数据库 – 整个过程都不需要修改代码。

V．构建，发布，运行

严格分离构建和运行

基准代码 转化为一份部署(非开发环境)需要以下三个阶段：

- *构建阶段* 是指将代码仓库转化为可执行包的过程。构建时会使用指定版本的代码，获取和打包 依赖项，编译成二进制文件和资源文件。
- *发布阶段* 会将构建的结果和当前部署所需 配置 相结合，并能够立刻在运行环境中投入使用。
- *运行阶段*（或者说“运行时”）是指针对选定的发布版本，在执行环境中启动一系列应用程序 进程。



12-facfor 应用严格区分构建，发布，运行这三个步骤。举例来说，直接修改处于运行状态的代码是非常不可取的做法，因为这些修改很难再同步回构建步骤。

部署工具通常都提供了发布管理工具，最引人注目的功能是退回至较旧的发布版本。比如， Capistrano 将所有发布版本都存储在一个叫 `releases` 的子目录中，当前的在线版本只需映射至对应的目录即可。该工具的 `rollback` 命令可以很容易地实现回退版本的功能。

每一个发布版本必须对应一个唯一的发布 ID，例如可以使用发布时的时间戳（`2011-04-06-20:32:17`），亦或是一个增长的数字（`v100`）。发布的版本就像

一本只能追加的账本，一旦发布就不可修改，任何的变动都应该产生一个新的发布版本。

新的代码在部署之前，需要开发人员触发构建操作。但是，运行阶段不一定需要人为触发，而是可以自动进行。如服务器重启，或是进程管理器重启了一个崩溃的进程。因此，运行阶段应该保持尽可能少的模块，这样假设半夜发生系统故障而开发人员又捉襟见肘也不会引起太大问题。构建阶段是可以相对复杂一些的，因为错误信息能够立刻展示在开发人员面前，从而得到妥善处理。

VI. 进程

以一个或多个无状态进程运行应用

运行环境中，应用程序通常是以一个和多个 *进程* 运行的。

最简单的场景中，代码是一个独立的脚本，运行环境是开发人员自己的笔记本电脑，进程由一条命令行（例如 `python my_script.py`）。另外一个极端情况是，复杂的应用可能会使用很多 进程类型 ，也就是零个或多个进程实例。

12-Factor 应用的进程必须无状态且 无共享 。 任何需要持久化的数据都要存储在 后端服务 内，比如数据库。

内存区域或磁盘空间可以作为进程在做某种事务型操作时的缓存，例如下载一个很大的文件，对其操作并将结果写入数据库的过程。12-Factor 应用根本不用考虑这些缓存的内容是不是可以保留给之后的请求来使用，这是因为应用启动了多种类型的进程，将来的请求多半会由其他进程来服务。即使在只有一个进程的情形下，先前保存的数据（内存或文件系统中）也会因为重启（如代码部署、配置更改、或运行环境将进程调度至另一个物理区域执行）而丢失。

源文件打包工具（Jammit, django-compressor） 使用文件系统来缓存编译过的源文件。12-Factor 应用更倾向于在 构建步骤做此动作——正如 Rails 资源管道 ，而不是在运行阶段。

一些互联网系统依赖于 “粘性 session”， 这是指将用户 session 中的数据缓存至某进程的内存中，并将同一用户的后续请求路由到同一个进程。粘性 session 是 12-Factor 极力反对的。Session 中的数据应该保存在诸如 Memcached 或 Redis 这样的带有过期时间的缓存中。

VII. 端口绑定

通过端口绑定(*Port binding*)来提供服务

互联网应用有时会运行于服务器的容器之中。例如 PHP 经常作为 Apache HTTPD 的一个模块来运行，正如 Java 运行于 Tomcat 。

12-Factor 应用完全自我加载 而不依赖于任何网络服务器就可以创建一个面向网络的服务。互联网应用 **通过端口绑定来提供服务**，并监听发送至该端口的请求。

本地环境中，开发人员通过类似 `http://localhost:5000/` 的地址来访问服务。

在线上环境中，请求统一发送至公共域名而后路由至绑定了端口的网络进程。

通常的实现思路是，将网络服务器类库通过 依赖声明 载入应用。例如，Python 的 Tornado, Ruby 的 Thin , Java 以及其他基于 JVM 语言的 Jetty。完全由 *用户端*，确切的说应该是应用的代码，发起请求。和运行环境约定好绑定的端口即可处理这些请求。

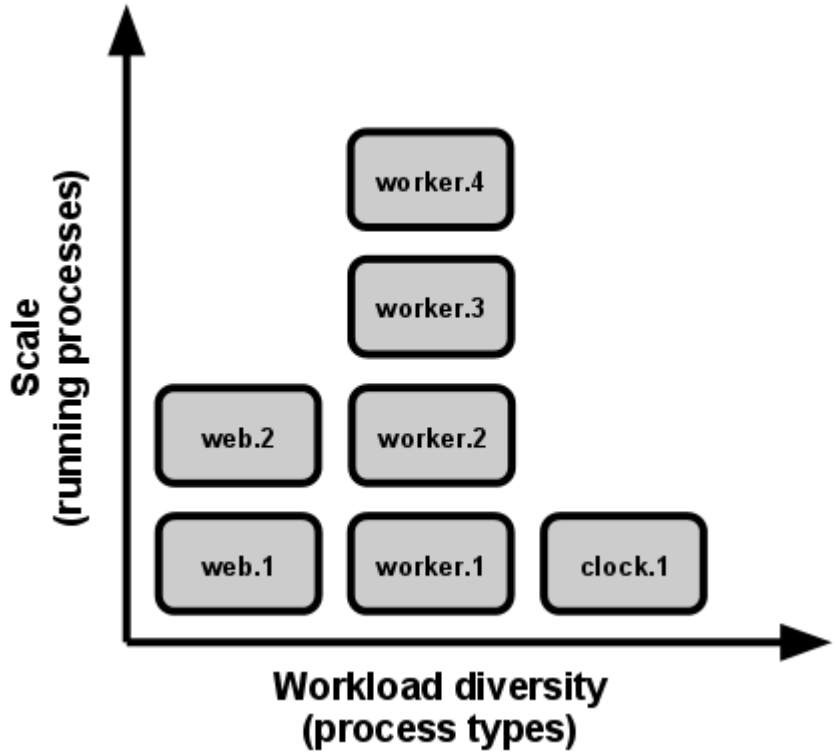
HTTP 并不是唯一一个可以由端口绑定提供的服务。其实几乎所有服务器软件都可以通过进程绑定端口来等待请求。例如，使用 XMPP 的 ejabberd， 以及使用 Redis 协议 的 Redis 。

还要指出的是，端口绑定这种方式也意味着一个应用可以成为另外一个应用的 后端服务 ，调用方将服务方提供的相应 URL 当作资源存入 配置 以备将来调用。

VIII. 并发

通过进程模型进行扩展

任何计算机程序，一旦启动，就会生成一个或多个进程。互联网应用采用多种进程运行方式。例如，PHP 进程作为 Apache 的子进程存在，随请求按需启动。Java 进程则采取了相反的方式，在程序启动之初 JVM 就提供了一个超级进程储备了大量的系统资源(CPU 和内存)，并通过多线程实现内部的并发管理。上述 2 个例子中，进程是开发人员可以操作的最小单位。



在 12-factor 应用中，进程是一等公民。12-Factor 应用的进程主要借鉴于 unix 守护进程模型 。开发人员可以运用这个模型去设计应用架构，将不同的工作分配给不同的 *进程类型* 。例如，HTTP 请求可以交给 web 进程来处理，而常驻的后台工作则交由 worker 进程负责。

这并不包括个别较为特殊的进程，例如通过虚拟机的线程处理并发的内部运算，或是使用诸如 EventMachine, Twisted,Node.js 的异步/事件触发模型。但一台独立的虚拟机的扩展有瓶颈（垂直扩展），所以应用程序必须可以在多台物理机器间跨进程工作。

上述进程模型会在系统急需扩展时大放异彩。12-Factor 应用的进程所具备的无共享，水平分区的特性 意味着添加并发会变得简单而稳妥。这些进程的类型以及每个类型中进程的数量就被称作 *进程构成* 。

12-Factor 应用的进程 不需要守护进程 或是写入 PID 文件。相反的，应该借助操作系统的进程管理器(例如 Upstart ， 分布式的进程管理云平台， 或是类似 Foreman 的工具)，来管理 输出流 ，响应崩溃的进程，以及处理用户触发的重启和关闭超级进程的请求。

IX. 易处理

快速启动和优雅终止可最大化健壮性

12-Factor 应用的 进程 是 *易处理（disposable）*的，意思是说它们可以瞬间开启或停止。这有利于快速、弹性的伸缩应用，迅速部署变化的 代码 或 配置，稳健的部署应用。

进程应当追求 **最小启动时间** 。 理想状态下，进程从敲下命令到真正启动并等待请求的时间应该只需很短的时间。更少的启动时间提供了更敏捷的 发布 以及扩展过程，此外还增加了健壮性，因为进程管理器可以在授权情形下容易的将进程搬到新的物理机器上。

进程 **一旦接收 终止信号（`SIGTERM`） 就会优雅的终止** 。就网络进程而言，优雅终止是指停止监听服务的端口，即拒绝所有新的请求，并继续执行当前已接收的请求，然后退出。此类型的进程所隐含的要求是 HTTP 请求大多都很短(不会超过几秒钟)，而在长时间轮询中，客户端在丢失连接后应该马上尝试重连。

对于 worker 进程来说，优雅终止是指将当前任务退回队列。例如，RabbitMQ 中，worker 可以发送一个 `NACK` 信号。Beanstalkd 中，任务终止并退回队列会在 worker 断开时自动触发。有锁机制的系统诸如 Delayed Job 则需要确定释放了系统资源。此类型的进程所隐含的要求是，任务都应该 可重复执行， 这主要由将结果包装进事务或是使重复操作 幂等 来实现。

进程还应当**在面对突然死亡时保持健壮**，例如底层硬件故障。虽然这种情况比起优雅终止来说少之又少，但终究有可能发生。一种推荐的方式是使用一个健壮的后端队列，例如 Beanstalkd ，它可以在客户端断开或超时后自动退回任务。无论如何，12-Factor 应用都应该可以设计能够应对意外的、不优雅的终结。Crash-only design 将这种概念转化为 合乎逻辑的理论。

X. 开发环境与线上环境等价

尽可能的保持开发，预发布，线上环境相同

从以往经验来看，开发环境（即开发人员的本地 部署）和线上环境（外部用户访问的真实部署）之间存在着很多差异。这些差异表现在以下三个方面：

- **时间差异：**开发人员正在编写的代码可能需要几天，几周，甚至几个月才会上线。
- **人员差异：** 开发人员编写代码，运维人员部署代码。
- **工具差异：** 开发人员或许使用 Nginx，SQLite，OS X，而线上环境使用 Apache，MySQL 以及 Linux。

12-Factor 应用想要做到 持续部署 就必须缩小本地与线上差异。 再回头看上面所描述的三个差异:

- 缩小时间差异：开发人员可以几小时，甚至几分钟就部署代码。
- 缩小人员差异：开发人员不只要编写代码，更应该密切参与部署过程以及代码在线上的表现。
- 缩小工具差异：尽量保证开发环境以及线上环境的一致性。

将上述总结变为一个表格如下：

	传统应用	12-Factor 应用
每次部署间隔	数周	几小时
开发人员 vs 运维人员	不同的人	相同的人
开发环境 vs 线上环境	不同	尽量接近

后端服务 是保持开发与线上等价的重要部分，例如数据库，队列系统，以及缓存。许多语言都提供了简化获取后端服务的类库，例如不同类型服务的 *适配器* 。下列表格提供了一些例子。

类型	语言	类库	适配器

数据库	Ruby/Rails	ActiveRecord	MySQL, PostgreSQL, SQLite
队列	Python/Django	Celery	RabbitMQ, Beanstalkd, Redis
缓存	Ruby/Rails	ActiveSupport::Cache	Memory, filesystem, Memcached

开发人员有时会觉得在本地环境中使用轻量的后端服务具有很强的吸引力，而那些更重量级的健壮的后端服务应该使用在生产环境。例如，本地使用 SQLite 线上使用 PostgreSQL；又如本地缓存在进程内存中而线上存入 Memcached。

12-Factor 应用的开发人员应该反对在不同环境间使用不同的后端服务，即使适配器已经可以几乎消除使用上的差异。这是因为，不同的后端服务意味着会突然出现的不兼容，从而导致测试、预发布都正常的代码在线上出现问题。这些错误会给持续部署带来阻力。从应用程序的生命周期来看，消除这种阻力需要花费很大的代价。

与此同时，轻量的本地服务也不像以前那样引人注目。借助于 Homebrew，apt-get 等现代的打包系统，诸如 Memcached、PostgreSQL、RabbitMQ 等后端服务的安装与运行也并不复杂。此外，使用类似 Chef 和 Puppet 的声明式配置工具，结合像 Vagrant 这样轻量的虚拟环境就可以使得开发人员的本地环境与线上环境无限接近。与同步环境和持续部署所带来的益处相比，安装这些系统显然是值得的。

不同后端服务的适配器仍然是有用的，因为它们可以使移植后端服务变得简单。但应用的所有部署，这其中包括开发、预发布以及线上环境，都应该使用同一个后端服务的相同版本。

XI. 日志

把日志当作事件流

日志 使得应用程序运行的动作变得透明。在基于服务器的环境中，日志通常被写在硬盘的一个文件里，但这只是一种输出格式。

日志应该是 事件流 的汇总，将所有运行中进程和后端服务的输出流按照时间顺序收集起来。尽管在回溯问题时可能需要看很多行，日志最原始的格式确实是一个事件一行。日志没有确定开始和结束，但随着应用在运行会持续的增加。

12-factor 应用本身从不考虑存储自己的输出流。 不应该试图去写或者管理日志文件。相反，每一个运行的进程都会直接的标准输出（`stdout`）事件流。开发环境中，开发人员可以通过这些数据流，实时在终端看到应用的活动。

在预发布或线上部署中，每个进程的输出流由运行环境截获，并将其他输出流整理在一起，然后一并发送给一个或多个最终的处理程序，用于查看或是长期存档。这些存档路径对于应用来说不可见也不可配置，而是完全交给程序的运行环境管理。类似 Logplex 和 Fluent 的开源工具可以达到这个目的。

这些事件流可以输出至文件，或者在终端实时观察。最重要的，输出流可以发送到 Splunk 这样的日志索引及分析系统，或 Hadoop/Hive 这样的通用数据存储系统。这些系统为查看应用的历史活动提供了强大而灵活的功能，包括：

- 找出过去一段时间特殊的事件。
- 图形化一个大规模的趋势，比如每分钟的请求量。
- 根据用户定义的条件实时触发警报，比如每分钟的报错超过某个警戒线。

XII. 管理进程

后台管理任务当作一次性进程运行

进程构成（process formation）是指用来处理应用的常规业务（比如处理 web 请求）的一组进程。与此不同，开发人员经常希望执行一些管理或维护应用的一次性任务，例如：

- 运行数据移植（Django 中的 `manage.py migrate`，Rails 中的 `rake db:migrate`）。
- 运行一个控制台（也被称为 REPL shell），来执行一些代码或是针对线上数据库做一些检查。大多数语言都通过解释器提供了一个 REPL 工具（`python` 或 `perl`），或是其他命令（Ruby 使用 `irb`，Rails 使用 `rails console`）。
- 运行一些提交到代码仓库的一次性脚本。

一次性管理进程应该和正常的 常驻进程 使用同样的环境。这些管理进程和任何其他的进程一样使用相同的 代码 和 配置，基于某个 发布版本 运行。后台管理代码应该随其他应用程序代码一起发布，从而避免同步问题。

所有进程类型应该使用同样的 依赖隔离 技术。例如，如果 Ruby 的 web 进程使用了命令 `bundle exec thin start`，那么数据库移植应使用 `bundle exec rake db:migrate`。同样的，如果一个 Python 程序使用了 Virtualenv，则需要在运行 Tornado Web 服务器和任何 `manage.py` 管理进程时引入 `bin/python`。

12-factor 尤其青睐那些提供了 REPL shell 的语言，因为那会让运行一次性脚本变得简单。在本地部署中，开发人员直接在命令行使用 shell 命令调用一次性管理进程。在线上部署中，开发人员依旧可以使用 ssh 或是运行环境提供的其他机制来运行这样的进程。