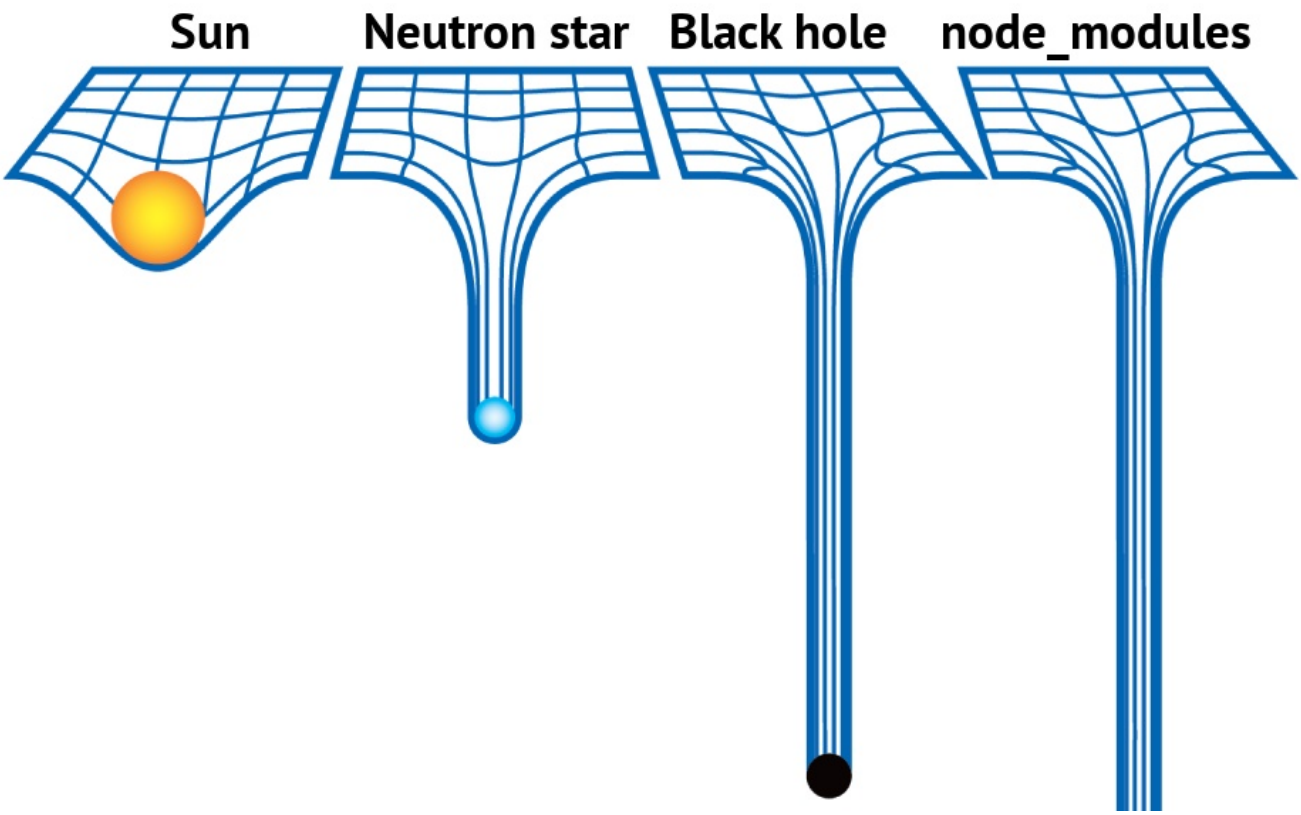
鱼肚的博客 Don't Repeat Yourself

2021-01-02

前端CI中的性能优化

前言

前端CI中的一个很难绕过的问题，是node_modules的处理。现在的前端生态建立在npm之上，而npm有黑洞之称，相信这个图大家都不陌生。



当然除了node_modules以外，还会有一些其它的性能瓶颈，这些问题都会逐渐地吞噬掉CI的时间，使的一次pipeline的时间越来越久，阻塞开发部署等流程。

常见的性能问题

在实际工作中，我遇到过一些影响性能的问题，这里和大家分享一下。

- node_modules的安装问题
- docker镜像拉取速度问题
- gitlab cache加载和保存缓慢问题
- runner性能跑满，服务器压力过高，导致整体变慢问题
- eslint等执行过慢问题

- 磁盘读写过慢问题
- 磁盘占用过多问题

常见的解决方案

一句话总结：缓存解万愁（当然缓存也带来新的烦恼，此处暂且不提）。

具体的策略包括但不限于如下范围：

- Gitlab CI中的Cache机制
- 镜像加速
- Gitlab Runner中docker executor的缓存
- Docker cache
- Yarn、npm 等cache
- ESLint Cache（针对eslint过慢的问题）
- BCache（混合SSD加速）
- 定期清理磁盘空间

实例分享与原理剖析

node_modules的优化

node_modules的体积比较大，涉及到的磁盘I/O和网络I/O比较多，再加上国内访问npm的速度不佳，很容易成为性能瓶颈。

对它的优化主要是如下几种思路：

- 最彻底的，能不安装就不安装。可使用基础镜像全局安装、Docker Cache等手段
- 次之的，在不得不安装node_modules的时候，缓存上一次的node_modules，或使用npm cache、yarn cache等。这种方式一般会用到 Gitlab CI中的cache机制
- 最后，切换镜像到一个国内镜像，也有助于加速。

先说尽量不安装node_modules的方法

全局安装

假定有一个工程，CI任务中只需要处理npm发包。那有可能node_modules中只有 typescript之类的工具。

如果是这种场景的话，可以考虑全局安装typescript等工具，代替项目中安装。

从原来的

```
FROM node:12-alpine
WORKDIR /workspace

ADD . /workspace
RUN npm i && npm run build && publish
```

转换成：

```
FROM node:12-alpine
WORKDIR /workspace
RUN npm i -g typescript

ADD . /workspace
RUN npm run build && publish
```

这样就省去了 `npm i` 这个过程，同时因为 `RUN npm i -g typescript` 这个步骤比较容易命中Docker cache，所以大多数时候并不会重复安装依赖。

全局安装的方法，很多时候还是避免不了安装node_modules，因为有可能用到一些更复杂的，必须随项目安装的依赖，比如一个需要webpack构建之后，部署构建产物的工程，也就是常见的前端业务工程，基本都实现不了全部用全局包。

这个时候就用到了依赖前置。

依赖前置

依赖前置是为了利用Docker Cache而对Dockerfile进行的一个改进。

修改之后的Dockerfile是这种样子：

```
FROM node:12-alpine
WORKDIR /workspace
RUN mkdir -p /workspace

COPY ./package.json ./package-lock.json /workspace
RUN npm i

ADD . /workspace
RUN npm run build && publish
```

它和上面的Dockerfile相比，主要的改动是把 npm install 放在了 `ADD . /workspace` 的前面。

按照Docker cache的原理，上面的layer如果没变，下面的 RUN 命令就会直接使用缓存。如果 `package.json` 和 `package-lock.json` 文件没有变化，就会命中缓存，如果这两个文件中任何一个有变化，则无法命中缓存，会重新执行安装。

在大多数场景中，发生变化的是代码，而非package.json等工程配置文件。因此 `ADD . /workspace` 之后基本上无法命中缓存，而 `COPY ./package.json ./package-lock.json /workspace` 之后则大概率能命中缓存。

以上两种都是避免安装node_modules实现加速，下面介绍下其它的方案：

Gitlab CI Cache

在有yarn cache或npm cache时，安装node_modules会比全新安装快不少，因为有不少的资源可以从本地获取了。

Gitlab CI中提供了一个Cache机制，可以指定要缓存的路径，在下次执行时先加载缓存，后执行任务。

比如说一个常见的CI配置：

```
cache: &global_cache
  key: ${CI_COMMIT_REF_SLUG}
  paths:
    - node_modules/
    - public/
    - vendor/
  policy: pull-push

job:
  cache:
    # inherit all global cache settings
    <<: *global_cache
    # override the policy
    policy: pull
```

具体情况，可直接查阅官方文档https://docs.gitlab.com/ee/ci/caching/。

在使用Gitlab CI Cache的时候需要注意，对于I/O压力比较高的服务器来说，CI Cache可能也会消耗较长的时间。

切换国内镜像

这一部分资源很多，此处不再赘述。可考虑使用nrm切换到taoabao等镜像源。

ESLint的优化

之前写过一篇博客专门讲这个：传送门

Gitlab Runner优化

Gitlab Runner中可以配置一些优化，我尝试过的主要是如下的方面：

- 通过限制资源和并发数，避免CI服务器限入激烈的资源竞争。
- 优先使用本地docker镜像，避免因为docker hub服务慢拖慢Runner

限制并发数

Gitlab Runner中可以设置并发数的上限，既有服务级别的，也有Job级别的。

示例配置：

```
concurrent = 7
check_interval = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "hello"
  limit = 2
  url = "http://git.mycompany.com/"
  token = "mytoken"
  executor = "docker"
  [runners.custom_build_dir]
  [runners.docker]
    tls_verify = false
    image = "docker"
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache", "/var/run/docker.sock:/var/run/docker.s
    pull_policy = "if-not-present"
    shm_size = 0

```

如上的示例代码中， `concurrent = 7` 代表此Runner服务最多同时跑7个Job。而 `limit = 2` 则代表 `hello` 这个job最多同时跑两个。

限制资源

限制资源是通过Docker提供的限制资源的方式，会需要较新的Docker服务版本。可以用来限制CPU、GPU、内存等资源。

如限制Docker run过程中可用的cpu资源：

```
1  docker run --cpus="1" xxx
```

详细使用方法参考官方文档：Runtime options with Memory, CPUs, and GPUs

优先使用本地镜像

Runner服务器的 `/etc/gitlab-runner/config.toml` 文件中，有关于docker executor的配置，在 `[[runners.docker]]` 中，默认是始终拉最新的，可以改成优先使用本地的：

```
1    [runners.docker]
2      pull_policy = "if-not-present"
```

配置成 `if-not-present` 之后，如果本地有此镜像，就不会再重新查询docker hub。

磁盘的优化

CI服务器需要的磁盘空间一般都比较大，尤其是在需要 `npm install` 的时候。所以有可能服务器中使用的是机械硬盘，而非固态。而读写速度对CI的影响会比较大。

在有些时候，有可能能有一些比较小的固态硬盘，CI存储全存在上面不太现实，但可以用来做混合SSD，实现I/O方面的优化。

Linux中可以使用BCache工具制作混合SSD，参考我之前写的博文：Linux中使用小容量SSD制作混合SSD硬盘。

定期清理磁盘空间

磁盘过满的时候，也会影响性能，当磁盘空间不足时，甚至会使CI服务直接停止工作。所以需要加一些清理脚本。

在使用Docker executor的时候，磁盘的占用主要是已经不再使用的Docker image，以及一些Docker volume。

要定期清理它们，可以使用下面的命令：

```bash
#!/bin/bash

# 清理镜像，保留一天
docker system prune -f --filter "until=$((1*24))h"

# 清理volume，保留三天
docker volume prune -f --filter "until=$((3*24))h"
```

把这个脚本随便起个名字，加上可执行权限，然后复制到 `/etc/cron.daily` 或 `/etc/cron.hourly` 等目录中，实现每天清理一次或每小时清理一次的效果。

以上。

twitter • github • stack overflow