Git

pwd：用于显示当前目录

git init: 把目录变成 git 可以管理的仓库

git 只能跟踪文本文件的改动，二进制文件只清楚改动了大小，而不能跟踪

git add <file> 告诉 git，把文件添加到仓库

git commit -m <message> 改善文件提交到仓库， -m 提交说明

git status 命令可以让我们时刻掌握仓库当前的状态

git diff <file> 查看文件具体修改了什么内容

git log 显示从最近到最远的提交日志 嫌输出信息太多 加上--pretty=oneline 参数

git reset 回退到某版本 `HEAD` 表示当前版本，上一个版本就是 `HEAD^` ，回退到后 100 版本 `HEAD~100`

git reflog 用来记录你的每一次命令

git checkout -- readme.txt 意思就是，把 readme.txt 文件在工作区的修改全部撤销

一种是 `readme.txt` 自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是 `readme.txt` 已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次 `git commit` 或 `git add` 时的状态。

添加到了暂存区，还没有提交 用命令 `git reset HEAD <file>` 可以把暂存区的修改撤销掉（unstage），重新放回工作区

小结

场景 1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令 `git checkout -- file`。

场景 2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令 `git reset HEAD <file>`，就回到了场景 1，第二步按场景 1 操作。

场景 3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退一节，不过前提是没有推送到远程库。

要从版本库中删除该文件，那就用命令 `git rm` 删掉，并且 `git commit`

另一种情况是删错了，因为版本库里还有呢 `git checkout -- test.txt` 把误删的文件恢复到最新版本

`git checkout` 其实是用版本库里的版本替换工作区的版本

本地仓库和远程仓库关联

```
git remote add origin git@github.com:michaelliao/learngit.git
```

`git push` 命令，实际上是把当前分支 `master` 推送到远程

```
git push -u origin master
```

第一次推送 `master` 分支时，加上了 `-u` 参数，Git 不但会把本地的 `master` 分支内容推送的远程新的 `master` 分支，还会把本地的 `master` 分支和远程的 `master` 分支关联起来，在以后的推送或者拉取时就可以简化命令。

`git clone url` 克隆到本地库

`git checkout` 命令加上 `-b` 参数表示创建并切换

`git branch` 命令会列出所有分支，当前分支前面会标一个 `*` 号

合并某分支到当前分支：`git merge <name>`

```
git branch -d 删除分支
```

切换分支这个动作，用 `switch` 更科学

```
git switch -c dev 创建并切换到新的 dev 分支
```

`git switch master` 直接切换到已有的 `master` 分支

`git merge --no-ff -m "merge with no-ff" dev` 准备合并 `dev` 分支，请注意`--no-ff` 参数，表示禁用 `Fast forward`

加上`--no-ff` 参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而 `fast forward` 合并就看不出来曾经做过合并。

`git stash` 可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作

`git stash list` 查看该分支下“储藏”起来的工作区

`git stash pop`，恢复的同时把 stash 内容也删了

`cherry-pick` 命令，让我们能复制一个特定的提交到当前分支

如果要丢弃一个没有被合并过的分支，可以通过 `git branch -D <name>` 强行删除。

多人协作的工作模式通常是这样：

首先，可以试图用 `git push origin <branch-name>` 推送自己的修改；

如果推送失败，则因为远程分支比你的本地更新，需要先用 `git pull` 试图合并；

如果合并有冲突，则解决冲突，并在本地提交；

没有冲突或者解决掉冲突后，再用 `git push origin <branch-name>` 推送就能成功！

如果 `git pull` 提示 `no tracking information`，则说明本地分支和远程分支的链接关系没有创建，用命令 `git branch --set-upstream-to <branch-name> origin/<branch-name>`。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

小结

查看远程库信息，使用 `git remote -v`；

本地新建的分支如果不推送到远程，对其他人就是不可见的；

从本地推送分支，使用`git push origin branch-name`，如果推送失败，先用`git pull`抓取远程的新提交；

在本地创建和远程分支对应的分支，使用`git checkout -b branch-name origin/branch-name`，本地和远程分支的名称最好一致；

建立本地分支和远程分支的关联，使用`git branch --set-upstream branch-name origin/branch-name`；

从远程抓取分支，使用`git pull`，如果有冲突，要先处理冲突。

Redis

Key 键

DEL：删除给定的一个或多个 key 。

String 字符串

GET：返回 key 所关联的字符串值。假如 key 储存的值不是字符串类型，返回一个错误

INCR:将 key 中储存的数字值增一。

如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 INCR 操作。

INCRBY:将 key 所储存的值加上增量 increment 。

MGET 返回所有(一个或多个)给定 key 的值。

MSET 同时设置一个或多个 key-value 对。

如果某个给定 key 已经存在，那么 MSET 会用新值覆盖原来的旧值

SET：将字符串值 value 关联到 key 。

如果 key 已经持有其他值， SET 就覆写旧值，无视类型。

Hash 哈希表

HDEL 删除哈希表 key 中的一个或多个指定域，不存在的域将被忽略。

HGET 返回哈希表 key 中给定域 field 的值。

HGETALL

返回哈希表 key 中，所有的域和值。

在返回值里，紧跟每个域名(field name)之后是域的值(value)，所以返回值的长度是哈希表大小的两倍。

HINCRBY

为哈希表 key 中的域 field 的值加上增量 increment 。

增量也可以为负数，相当于对给定域进行减法操作。

如果 key 不存在，一个新的哈希表被创建并执行 HINCRBY 命令。

如果域 field 不存在，那么在执行命令前，域的值被初始化为 0 。

对一个储存字符串值的域 field 执行 HINCRBY 命令将造成一个错误。

本操作的值被限制在 64 位(bit)有符号数字表示之内。

HMGET

返回哈希表 key 中，一个或多个给定域的值。

如果给定的域不存在于哈希表，那么返回一个 nil 值。

因为不存在的 key 被当作一个空哈希表来处理，所以对一个不存在的 key 进行 *HMGET* 操作将返回一个只带有 nil 值的表。

HMSET

HSET 将哈希表 key 中的域 field 的值设为 value 。

如果 key 不存在，一个新的哈希表被创建并进行 HSET 操作。

如果域 field 已经存在于哈希表中，旧值将被覆盖。

LIST

LPOP

移除并返回列表 key 的头元素。

LPUSH

将一个或多个值 value 插入到列表 key 的表头

如果有多个 value 值，那么各个 value 值按从左到右的顺序依次插入到表头： 比如说，对空列表 mylist 执行命令 LPUSH mylist a b c ，列表的值将是 c b a ，这等同于原子性地执行 LPUSH mylist a 、 LPUSH mylist b 和 LPUSH mylist c 三个命令。

如果 key 不存在，一个空列表会被创建并执行 *LPUSH* 操作。

当 key 存在但不是列表类型时，返回一个错误。

LRANGE

返回列表 key 中指定区间内的元素，区间以偏移量 start 和 stop 指定。

下标(index)参数 start 和 stop 都以 0 为底，也就是说，以 0 表示列表的第一个元素，以 1 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 -1 表示列表的最后一个元素， -2 表示列表的倒数第二个元素，以此类推。

LREM

LREM key count value

根据参数 count 的值，移除列表中与参数 value 相等的元素。

count 的值可以是以下几种：

count > 0 : 从表头开始向表尾搜索，移除与 value 相等的元素，数量为 count 。

count < 0 : 从表尾开始向表头搜索，移除与 value 相等的元素，数量为 count 的绝对值。

count = 0 : 移除表中所有与 value 相等的值。

LSET

LSET key index value

将列表 key 下标为 index 的元素的值设置为 value 。

当 index 参数超出范围，或对一个空列表(key 不存在)进行 LSET 时，返回一个错误。

RPOP

RPOP key

移除并返回列表 key 的尾元素。

RPUSH

RPUSH key value [value ...]

将一个或多个值 value 插入到列表 key 的表尾(最右边)。

如果有多个 value 值，那么各个 value 值按从左到右的顺序依次插入到表尾：比如对一个空列表 mylist 执行 RPUSH mylist a b c ，得出的结果列表为 a b c ，等同于执行命令 RPUSH mylist a 、 RPUSH mylist b 、 RPUSH mylist c 。

如果 key 不存在，一个空列表会被创建并执行 *RPUSH* 操作。

当 key 存在但不是列表类型时，返回一个错误。

Set 集合

SADD

SCARD

SISMEMBER

SMEMBERS

SREM

SortedSet 有序集合

ZADD

ZCARD

ZCOUNT

ZRANGE

ZRANGEBYSCORE

ZRANK

ZREM

ZREMRANGEBYRANK

ZREMRANGE

ZREVRANGEBYSCORE

ZREVRANK

ZSCORE

微服务

https://blog.csdn.net/zmbaliqq/article/details/84936551

所谓服务，一定要区别于系统，服务一个或者一组相对较小且独立的功能单元，是用户可以感知最小功能集。

微，狭义来讲就是体积小

微服务架构风格是一种使用一套小服务来开发单个应用的方式途径，每个服务运行在自己的进程中，并使用轻量级机制通信，通常是 HTTP API，这些服务基于业务能力构建，并能够通过自动化部署机制来独立部署，这些服务使用不同的编程语言实现，以及不同数据存储技术，并保持最低限度的集中式管理。

微服务与单体架构区别

单体架构所有的模块全都耦合在一块，代码量大，维护困难，微服务每个模块就相当于一个单独的项目，代码量明显减少，遇到问题也相对来说比较好解决。

单体架构所有的模块都共用一个数据库，存储方式比较单一，微服务每个模块都可以使用不同的存储方式（比如有的用 redis，有的用 mysql 等），数据库也是单个模块对应自己的数据库。

单体架构所有的模块开发所使用的技术一样，微服务每个模块都可以使用不同的开发技术，开发模式更灵活。

微服务与 SOA 区别

微服务，从本质意义上看，还是 SOA 架构。但内涵有所不同，微服务并不绑定某种特殊的技术，在一个微服务的系统中，可以有 Java 编写的服务，也可以有 Python 编写的服务，他们是靠 Restful 架构风格统一成一个系统的。所以微服务本身与具体技术实现无关，扩展性强。

SOA

面向服务架构，它可以根据需求通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用。服务层是 SOA 的基础，可以直接被应用调用，从而有效控制系统中与软件代理交互的人为依赖性。SOA 是一种粗粒度、 松耦合

服务架构，服务之间通过简单、精确定义接口进行通讯，不涉及底层编程接口和通讯模型。

微服务

简单来说微服务架构是采用一组服务 的方式来构建一个应用，服务独立部署在不同的进程中，不同服务通过一些轻 量级交互机制来通信,例如 RPC、HTTP 等，服务可独立扩展伸缩，每个服务定义了明确的边界，不同的服务甚至可以采用不同的编程语言来实现，由独立的团队来维护。

微服务设计原则

单一职责原则

意思是每个微服务只需要实现自己的业务逻辑就可以了，比如订单管理模块，它只需要处理订单的业务逻辑就可以了，其它的不必考虑。

服务自治原则

意思是每个微服务从开发、测试、运维等都是独立的，包括存储的数据库也都是独立的，自己就有一套完整的流程，我们完全可以把它当成一个项目来对待。不必依赖于其它模块。

轻量级通信原则

首先是通信的语言非常的轻量，第二，该通信方式需要是跨语言、跨平台的，之所以要跨平台、跨语言就是为了让每个微服务都有足够的独立性，可以不受技术的钳制。

接口明确原则

由于微服务之间可能存在着调用关系，为了尽量避免以后由于某个微服务的接口变化而导致其它微服务都做调整，在设计之初就要考虑到所有情况，让接口尽量做的更通用，更灵活，从而尽量避免其它模块也做调整。

微服务优势与缺点

7.1 特性

每个微服务可独立运行在自己的进程里；

一系列独立运行的微服务共同构建起了整个系统；

每个服务为独立的业务开发，一个微服务一般完成某个特定的功能，比如：订单管理，用户管理等；

微服务之间通过一些轻量级的通信机制进行通信，例如通过 REST API 或者 RPC 的方式进行调用。

注册中心

服务之间需要创建一种服务发现机制，用于帮助服务之间互相感知彼此的存在。服务启动时会将自身的服务信息注册到注册中心，并订阅自己需要消费的服务。

服务注册中心是服务发现的核心。它保存了各个可用服务实例的网络地址（IPAddress 和 Port）。服务注册中心必须要有高可用性和实时更新功能。上面提到的 Netflix Eureka 就是一个服务注册中心。它提供了服务注册和查询服务信息的 REST API。服务通过使用 POST 请求注册自己的 IPAddress 和 Port。每 30 秒发送一个 PUT 请求刷新注册信息。通过 DELETE 请求注销服务。客户端通过 GET 请求获取可用的服务实例信息。

负载均衡

服务高可用的保证手段，为了保证高可用，每一个微服务都需要部署多个服务实例来提供服务。此时客户端进行服务的负载均衡。

3.1.1 随机

把来自网络的请求随机分配给内部中的多个服务器。

3.1.2 轮询

每一个来自网络中的请求，轮流分配给内部的服务器，从 1 到 N 然后重新开始。此种负载均衡算法适合服务器组内部的服务器都具有相同的配置并且平均服务请求相对均衡的情况。

3.1.3 加权轮询

根据服务器的不同处理能力，给每个服务器分配不同的权值，使其能够接受相应权值数的服务请求。例如：服务器 A 的权值被设计成 1，B 的权值是 3，C

的权值是 6，则服务器 A、B、C 将分别接受到 10%、30%、60%的服务请求。此种均衡算法能确保高性能的服务器得到更多的使用率，避免低性能的服务器负载过重。

3.1.4 IP Hash

这种方式通过生成请求源 IP 的哈希值，并通过这个哈希值来找到正确的真实服务器。这意味着对于同一主机来说他对应的服务器总是相同。使用这种方式，你不需要保存任何源 IP。但是需要注意，这种方式可能导致服务器负载不平衡。

3.1.5 最少连接数

客户端的每一次请求服务在服务器停留的时间可能会有较大的差异，随着工作时间加长，如果采用简单的轮循或随机均衡算法，每一台服务器上的连接进程可能会产生极大的不同，并没有达到真正的负载均衡。最少连接数均衡算法对内部中需负载的每一台服务器都有一个数据记录，记录当前该服务器正在处理的连接数量，当有新的服务连接请求时，将把当前请求分配给连接数最少的服务器，使均衡更加符合实际情况，负载更加均衡。此种均衡算法适合长时处理的请求服务，如 FTP。

高可用

一般来说，我们的观念里一个服务至少要做到 99.9% 才称为基本上可用，是合格性产品。否则基本很难被别人使用。

MTBF 和 MTTR

MTBF: Mean time between Failures，用通俗的话讲，就是一个东西有多不可靠，多长时间坏一次。

MTTR: Mean time to recover，意思就是一旦坏了，恢复服务的时间需要多长。

一个服务的可用度，取决于 MTBF 和 MTTR 这两个因子。那就是：要么提高 MTBF，要么降低 MTTR

高可用性指你提供的服务要始终可用，不管天灾(停电，断网，磁盘空间满，服务器硬件损坏等)，人祸(软件 bug，黑客破坏，误操作等)，甚至地震，洪水抑或战争.

高可用性设计

高可用的不二法宝是冗余，也就是说，为了避免单点失败(Single Point Failure)，会增加一到多个点，而且最好放在不同的物理位置，降低多点失败的概率.

具体来说，服务器之间的关系有主从(primary/slave)关系，主主(active-active)关系.

再细分的话有一主多仆，多主多仆，对等自治等关系，是一个服务器集集群还是多个服务器集群.

过载保护

所谓的过载就是超出了服务本身的服务能力。达到服务运行的瓶颈了，用户体验已经崩溃了

过载保护有两个重要的点，一个是不要被别人拖垮了。一个是认怂。

过载保护的策略

前面介绍了一些过载预防和过载保护的具体方法，就过载保护而言，主要是使用了以下策略：

- **快速返回策略：** 利用监控告警的，client 获取 server 状态，如果认定其不可用，则不需要再把请求发给 Server。请求快速返回。
- **快速拒绝策略：** Server 根据自身状态，拒绝掉一些请求。

做资源隔离，禁止某个请求占用大量的线程资源。

接口随机的进行直接返回操作

过载保护其实并不是什么神奇的事情，主要应对上方流量突增的情况。这里过载保护一般只是少量接口，突然爆发大量请求。后面会有其他的部分问题讲解，比如下层出现调用超时，保证自己的服务可用。会提及服务降级和熔断策略。其实无论是过载保护还是熔断降级都是为了服务的高可用可用性考虑的。

序列化/反序列化

1.序列化是指把对象转换为字节序列的过程，而反序列化是指把字节序列恢复为对象的过程

2.对象序列化的最主要的用处就是在传递和保存对象的时候，保证对象的完整性和可传递性。序列化是把对象转换成有序字节流，以便在网络上传输或者保存在本地文件中。

3.序列化机制的核心作用就是对象状态的保存与重建。

4.反序列化就是客户端从文件中或网络上获得序列化后的对象字节流后，根据字节流中所保存的对象状态及描述信息，通过反序列化重建对象。

5.序列化就是把实体对象状态按照一定的格式写入到有序字节流，反序列化就是从有序字节流重建对象，恢复对象状态。

集中式日志

日志主要包括系统日志、应用程序日志（业务日志）以及安全日志。系统运维和开发人员通过日志了解服务器软硬件信息、查阅日志信息以及分析错误发生的原因等。

随着系统的日益复杂，大数据时代的来临，越来越容易就涉及到几十甚至上百台的服务器，因此迫切需要有一套针对日志的集中式管理平台产品。我们通过 ELK 实现了集中式日志管理平台，该平台统一涵盖了分布式日志收集、检索、统计、分析以及对日志信息的 Web 管理等集中化管控。

配置中心

随着业务的发展、微服务架构的升级，服务的数量、程序的配置日益增多（各种微服务、各种服务器地址、各种参数），传统的配置文件方式和数据库的方式已无法满足开发人员对配置管理的要求：

安全性：配置跟随源代码保存在代码库中，容易造成配置泄漏

时效性：修改配置，需要重启服务才能生效

局限性：无法支持动态调整：例如日志开关、功能开关

因此，我们需要配置中心来统一管理配置。

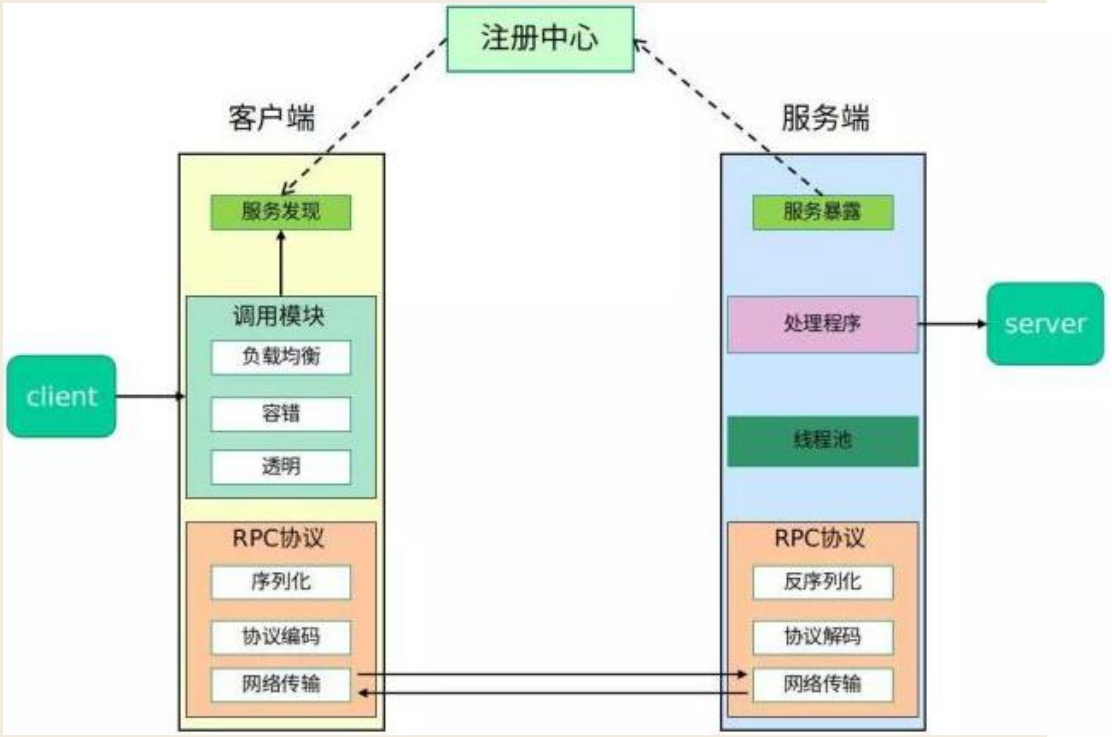
本文采取的配置中心为 Apollo，携程框架部门研发的开源配置管理中心，能够集中化管理应用不同环境，不同集群的配置，配置修改后能够实时推送到应用端，并且具备规范的权限、流程治理等特性。

RPC

RPC(Remote Procedure Call)：远程过程调用，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的思想。

RPC 是一种技术思想而非一种规范或协议

在一个典型 RPC 的使用场景中，包含了服务发现、负载、容错、网络传输、序列化等组件，其中“RPC 协议”就指明了程序如何进行网络传输和序列化。



RPC 要解决的两个问题：

1. 解决分布式系统中，服务之间的调用问题。
2. 远程调用时，要能够像本地调用一样方便，让调用者感知不到远程调用的逻辑。

可伸缩性

可伸缩性(可扩展性)是一种对软件系统计算处理能力的设计指标，高可伸缩性代表一种弹性，在系统扩展成长过程中，软件能够保证旺盛的生命力，通过很少的改动甚至只是硬件设备的添置，就能实现整个

系统处理能力的线性增长，实现高吞吐量和低延迟高性能。

低延迟，也就是用户能感受到的系统响应时间，比如一个网页在几秒内打开，越短表示延迟越低

吞吐量表示同时有多少用户能够享受到这种低延迟，

扩展性的目标是用可接受的延迟获得最大的吞吐量。可靠性(可用性)目标：用可接受的延迟获得数据更新的一致性。

跨语言调用

每个业务线有各自开发语言的选择权，便出现了 nodejs，python，go 多语言调用的问题。

解决跨语言调用的思路无非是两种：

寻找一个通用的协议

使用 agent 完成协议的适配

dubbo2.js 是 千米网 贡献给 dubbo 社区的一款 nodejs dubbo 客户端，它提供了 nodejs 对原生 dubbo 协议的支持，使得 nodejs 和 java 这两种异构语言的 rpc 调用变得便捷，高效。

守护进程

Daemon 是长时间运行的进程，通常在系统启动后就运行，在系统关闭是才结束。一般说 Daemon 程序在后台运行，是因为它没有控制终端，无法和前台的用户交互。Daemon 程序一般都作为服务程序使用，等待客户端程序与它通信。我们把运行的 Daemon 程序称作守护进程。

Linux Daemon（守护进程）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。它不需要用户输入

就能运行而且提供某种服务，不是对整个系统就是对某个用户程序提供服务。Linux 系统的大多数服务器就是通过守护进程实现的。常见的守护进程包括系统日志进程 syslogd、 web 服务器 httpd、邮件服务器 sendmail 和数据库服务器 mysqld 等。

守护进程一般在系统启动时开始运行，除非强行终止，否则直到系统关机都保持运行。守护进程经常以超级用户（root）权限运行，因为它们要使用特殊的端口（1-1024）或访问某些特殊的资源。

一个守护进程的父进程是 init 进程，因为它真正的父进程在 fork 出子进程后就先于子进程 exit 退出了，所以它是一个由 init 继承的孤儿进程。守护进程是非交互式程序，没有控制终端，所以任何输出，无论是向标准输出设备 stdout 还是标准出错设备 stderr 的输出都需要特殊处理。

守护进程的名称通常以 d 结尾，比如 sshd、xinetd、crond 等

IP

IP 是 Internet Protocol（网际互连协议）的缩写，是 TCP/IP 体系中的网络层协议。设计 IP 的目的是提高网络的可扩展性：一是解决互联网问题，实现大规模、异构网络的互联互通；二是分割顶层网络应用和底层网络技术之间的耦合关系，以利于两者的独立发展。根据端到端的设计原则，IP 只为主机提供一种无连接、不可靠的、尽力而为的数据报传输服务。

IP 位于 TCP/IP 模型的网络层(相当于 OSI 模型的网络层)，对上可载送传输层各种协议的信息，例如 TCP、UDP 等；对下可将 IP 信息包放到链路层，通过以太网、令牌环网络

等各种技术来传送。

端口

"端口"是英文 port 的意译，可以认为是设备与外界通讯交流的出口。端口可分为虚拟端口和物理端口，其中虚拟端口指计算机内部或交换机路由器内的端口，不可见。例如计算机中的 80 端口、21 端口、23 端口等。物理端口又称为接口，是可见端口，计算机背板的 RJ45 网口，交换机路由器集线器等 RJ45 端口。电话使用 RJ11 插口也属于物理端口的范畴。

超时

超时即当网络设备想在某个特定时间内从另一网络设备上接

收信息，但是失败的情况。其结果常为：重新传输信息或解除两设备间的会话。

心跳

定期检测服务器是否在线

就是每隔几分钟发送一个固定信息给服务端，服务端收到后回复一个固定信息如果服务端几分钟内没有收到客户端信息则视客户端断开。

发包方：可以是客户也可以是服务端，看哪边实现方便合理。 心跳包之所以叫心跳包是因为：它像心跳一样每隔固定时间发一次，以此来告诉服务器，这个客户端还活着。事实上这是为了保持长连接，至于这个包的内容，是没有什么特别规定的，不过一般都是很小的包，或者只包含包头的一个空包。心跳包主要也就是用于长连接的保活和断线处理。一般的应用下，判定时间在 30-40 秒比较不错。如果实在要求高，那就在 6-9 秒。

心跳不是服务端主动去发信息检测客户端状态，而是在服务端保存下来所有客户端的状态信息，然后等待客户端定时来访问服务端，更新自己的当前状态，如果客户端超过指定的时间没有来更新状态，则认为客户端已经宕机或者其状态异常。

异步队列

js 中异步队列可以分为两类,marcotask 队列和 microtask 队列, marcotask 队列里面包含有 1.script 中的所有同步代码,2.setTimeout,3. setInterval,4.setImmediate5. I/O 操作, 6. UI 渲染,而 microtask 队列主要是有 1.process,2.nextTick, 3.promise 等等， 异步队列执行的顺序主要是 先从 marcotask 中取出一个任务(第一次就是取出所有同步的代码),执行完成之后从 microtask 队列取出所有的任务,执行完成之后，再从 marcotask 队列中取出一个，循环往复,直到所有队列的任务被完成.

异地容灾

异地容灾，顾名思义就是在不同的地域，构建一套或者多套相同的应用或者数据库，起到灾难后立刻接管的作用。

异地容灾对企业应用及数据库起到了安全性、业务连续性等方面的作用

异地备份，通过互联网 TCP/IP 协议，备特佳容灾备份系统将本地的数据实时备份到异地服务器中，可以通过异地备份的数据进行远程恢复，也可以在异地进行数据回退，异地备份，如果想做接管需要专线连接，只有在同一网段内才能实现业务的接管。

乐观锁

每次获取数据的时候，都不会担心数据被修改，所以每次获取数据的时候都不会进行加锁，但是在更新数据的时候需要判断该数据是否被别人修改过。如果数据被其他线程修改，则不进行数据更新，如果数据没有被其他线程修改，则进行数据更新。由于数据没有进行加锁，期间该数据可以被其他线程进行读写操作。一般使用 version 方式和 CAS 操作方式。

比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。

TPS/QPS

TPS：Transactions Per Second（每秒传输的事物处理个数），即服务器每秒处理的事务数。TPS 包括一条消息入和一条消息出，加上一次用户数据库访问。（业务 TPS = CAPS × 每个呼叫平均 TPS）

TPS 是软件测试结果的测量单位。一个事务是指一个客户机向服务器发送请求然后服务器做出反应的过程。客户机在发送请求时开始计时，收到服务器响应后结束计时，以此来计算使用的时间和完成的事务个数。

一般的，评价系统性能均以每秒钟完成的技术交易的数量来衡量。系统整体处理能力取决于处理能力最低模块的 TPS 值。

二、QPS：每秒查询率，QPS 是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准，在因特网上，作为域名系统服务器的机器的性能经常用每秒查询率来衡量。

对应 fetches/sec，即每秒的响应请求数，也即是最大吞吐能力。

一次页面请求，可能产生多次对服务器的请求，服务器对这些请求，就可计入“Qps”之中。

例如：访问一个页面会请求服务器 3 次，一次放，产生一个“T”，产生 3 个“Q”

冷备

冷备是指两个服务器，一台运行，一台不运行做为备份。这样一旦运行的服务器宕机，就把备份的服务器运行起来。冷备的方案比较容易实现，但冷备的缺点是主机出现故障时备机不会自动接管，需要主动切换服务。冷备时，仍然是主数据中心担任用户的业务服务，但是冷备情况时，备份中心是不会对主数据中心进行实时性的备份，也就是说如果主数据中心出现故障，用户的业务也就中断了。

双活

双活容灾即灾备系统中使主生产端数据库和备机端数据库同时在线运行，处于可读可查询的状态的技术。

双活数据中心主要的目的是让主备数据中心一同工作，既不浪费资源，又会让用户的业务不被轻易的中断，工作时仍然是主数据中心负载多点，备数据中心起到防御的工作，占少一半。

回滚

回滚（Rollback）指的是程序或数据处理错误，将程序或数据恢复到上一次正确状态的行为。回滚包括程序回滚和数据回滚等类型。

泛型

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参

数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

服务降级

当服务器压力剧增的情况下，根据实际业务情况及流量，对一些服务和页面有策略的不处理或换种简单的方式处理，从而释放服务器资源以保证核心交易正常运作或高效运作。

服务降级主要用于什么场景呢？当整个微服务架构整体的负载超出了预设的上限阈值或即将到来的流量预计将会超过预设的阈值时，为了保证重要或基本的服务能正常运行，我们可以将一些 **不重要** 或 **不紧急** 的服务或任务进行服务的 **延迟使用** 或 **暂停使用。**

消息队列

“消息”是在两台计算机间传送的数据单位。消息可以非常简单，例如只包含文本字符串；也可以更复杂，可能包含嵌入对象。

消息被发送到队列中。“消息队列”是在消息的传输过程中保存消息的容器。消息队列管理器在将消息从它的源中继到它的目标时充当中间人。队列的主要目的是提供路由并保证消息的传递；如果发送消息时接收者不可用，消息队列会保留消息，直到可以成功地传递它。

消息队列是分布式系统中重要的组件，使用消息队列主要是为了通过异步处理提高系统性能和削峰、降低系统耦合性。

1. 通过异步处理提高系统性能（削峰、减少响应所需时间）;2.降低系统耦合性。

不使用消息队列服务器的时候，用户的请求数据直接写入数据库，在高并发的情况下数据库压力剧增，使得响应速度变慢。但是在使用消息队列之后，用户的请求数据发送给消息队列之后立即 返回，再由消息队列的消费者进程从消息队列中获取数据,异步写入数据库。由于消息队列服务器处理速度快于数据库(消息队列也比数据库有更好的伸缩性），因此响应速度得到大幅改善。

染色日志

每次记日志的时候，都打把 seqence id 打出来，rpc 调用的时候，由于跨机器了，需要在 rpc 消息中把这个 seqence id 传过去，rpc 服务器接收到消息，同时打印的所有日志也都带着这个 seqence id。这样所有的服务（不管你有多少台服务器）处理同一个请求的时候，就会打印出相同的 seqence id，再由一个统一的服务去收集这些日志，把相同 id 的日志收集到一起，按时间排序，汇总给程序员看，就能看到一次请求发起后，所有的 rpc 服务器打印的日志了。