本文内容出自：https://github.com/gzc426/Java-Interview

以后有更新内容，会在 github 更新

加入**互联网 IT** 求职、技术交流、资料共享 QQ 群 **691206173**，3T 编程资料等你来拿，

群主（群号：**691206173**）：本硕就读于**哈尔滨工业大学**，计算机专业，2019 硕士毕业，已拿**百度** java 后台开发 offer，另外还有去哪儿，华为，茄子快传，vipkid,秒针，创新工厂一点资讯这些公司的 offer。公众号中的文章有分享群主找工作的经验，java 学习/C++学习/机器学习/前端的指导路线，以及未来在百度的学习成长之路，满满都是干货，除了干货分享还有 3T 编程资料**（java/C++/算法/php/机器学习/大数据/人工智能/面试等）**等你来拿，另外还有微信交流群以及群主的**个人微信（抽空提供一对一指导意见）,当然也希望你能帮忙在朋友圈转发推广一下**

公众号

Nginx

Nginx 配置（如果项目使用到了的话），负载均衡机制 ；nginx 的请求转发算法，如何配置

根据权重转发

Nginx 是一个用 C 语言开发的高性能 WEB 服务器及反向代理服务器。
Nginx 底层使用 epoll，使用反应器模式。主**事件循环**等待操作系统发出准备事件的信号，这样数据就可以从套接字读取，在该实例中读取到缓冲区并进行处理。单个线程可以提供数万个并发连接。

直接使用 c/c++进行二次开发，对于很多用户是有一定门槛的，且 c/c++的开发效率也比不上 python、js、lua 等语言，python、js、lua 三者中，lua 是解析器最小，性能最高的语言，luajit 比 lua 又快数 10 倍。目前将 nginx 和 lua 结合在一起的有春哥维护的 openresty 和淘宝维护的 Tengine。
OpenResty 是一个基于 Nginx 与 Lua 的高性能 Web 平台，其内部集成了大量精良的 Lua 库、第三方模块以及大多数的依赖项。用于方便地搭建能够处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关。

OpenResty 通过汇聚各种设计精良的 Nginx 模块（主要由 OpenResty 团队自主开发），从而将 Nginx 有效地变成一个强大的通用 Web 应用平台。这样，Web 开发人员和系统工程师可以使用 Lua 脚本语言调动 Nginx 支持的各种 C 以及 Lua 模块，快速构造出足以胜任 10K 乃至 1000K 以上单机并发连接的高性能 Web 应用系统。

Nginx 目前提供的负载均衡模块：
ngx_http_upstream_round_robin，加权轮询，可均分请求，是默认的 HTTP 负载均衡算法，集成在框架中。
ngx_http_upstream_ip_hash_module，IP 哈希，可保持会话。
ngx_http_upstream_least_conn_module，最少连接数，可均分连接。
ngx_http_upstream_hash_module，一致性哈希，可减少缓存数据的失效。

Log4J/Slf4J

日志框架主要分两种：

一种是日志框架门面(facade)，用来统一一个共同的接口，方便我们用相同的代码支持不同的实现方法。主要包括 commons-logging 和 slf4j 两套规范。
另一种就是日志框架的实现了，主要包括 log4j、log4j2、logback、slf4j-simple 和 java.util.logging 包。
Slf4J：The Simple Logging Facade for Java
slf4j 译为简单日志门面，是日志框架的抽象。而 log4j 和 logback 是众多日志框架中的几种。
log4j 是 apache 实现的一个开源日志组件。(Wrapped implementations)
logback 同样是由 log4j 的作者设计完成的，拥有更好的特性，用来取代 log4j 的一个日志框架。是 slf4j 的原生实现。
logback 是直接实现了 slf4j 的接口，是不消耗内存和计算开销的。而 log4j 不是对 slf4j 的原生实现，所以 slf4j api 在调用 log4j 时需要一个适配层。

总结：
slf4j 是 java 的一个日志门面，实现了日志框架一些通用的 api，log4j 和 logback 是具体的日志框架。
他们可以单独的使用，也可以绑定 slf4j 一起使用。
单独使用。分别调用框架自己的方法来输出日志信息。
绑定 slf4j 一起使用。调用 slf4j 的 api 来输入日志信息，具体使用与底层日志框架无关（需要底层框架的配置文件）

SPI

SPI 就是(Service Provider Interface)，字面意思就是服务提供接口。
服务调用者跟服务提供者之间商定了一个协议：

服务发现者需要定义一个接口。
服务提供者要实现之前的接口，然后在 classpath 里的 **META-INF/services** 文件夹下新建一个文件，文件名是之前的接口的全类名，文件内容是实现类的全类名。
服务发现者保证会通过 ServiceLoader 在类路径内的所有 jar 包中搜索指定接口的实现类，进行实例化。
显然，一般来讲服务发现者一般就不能直接通过构造函数来构造这个接口的实现类，而是通过静态工厂方式封装实例化的过程。
ServiceLoader<AInterface> loader = ServiceLoader.load(AInterface.class);
ServiceLoader 会去类路径中查找所有支持了 AInterface 接口的实现类，并返回一个迭代器，这个迭代器会实例化所有的实现类。

Dubbo

Dubbo 通信原理；序列化原理；超时重试；负载均衡；

RocketMQ

RocketMQ 原理 可靠消息原理 尝试看一下源码；优点

Druid

Druid 原理，数据源/数据库连接池原理；连接池中的连接是长连接还是短连接？为什么？；连接池中的连接是基于什么协议的连接？为什么？
数据库连接池； mysql 数据库连接池的驱动参数 ；数据库连接池如何防止失效；数据库连接池代码

Zookeeper（分布式数据主备系统）

Zookeeper 的 Leader 选举过程，实现机制，有缓存，如何存储注册服务的；事务，结点，服务提供方挂了如何告知消费方；3 个节点挂掉一个能正常工作吗？

简介

Zookeeper 为分布式应用 提供了高效且可靠的分布式协调服务，提供了诸如**统一命名服务、发布订阅、负载均衡、配置管理和分布式锁**等分布式的基础服务。
设计目标是将那些复杂且容易出错的分布式一致性服务封装起来，构成一个高效可靠的原语集，并以一系列简单易用的接口提供给用户使用。
可以保证以下特性：

- 顺序一致性：从同一个客户端发起的事务请求，最终将会严格地按照其发起顺序被应用到 Zookeeper 中
- 原子性：要么集群中所有机器都成功应用了某一个事务，要么都没有应用。
- 单一视图：无论客户端连接的是哪一个 Zookeeper 服务器，其看到的服务端数据模型都是一致性的
- 可靠性：一旦服务端成功应用了某个事务，并完成对客户端的响应，那么该事务所引起的服务端状态变更将会被一直保留下来。
- 实时性：Zookeeper 能保证在一定的时间段内，客户端最终一定能够从服务端上读取到最新的数据状态。

基本概念/原理

集群角色

通常在分布式系统中，构成一个集群的每一台机器都有自己的角色，最典型的集群模式就是 Master/Slave 模式。而在 Zookeeper 中，引入了 Leader、Follower、Observer（可以没有）三种角色。Zookeeper 集群中的所有机器通过一个 Leader 选举过程选定一台被称为“Leader”的机器，Leader 服务器为客户端提供读和写服务。Follower 和 Observer 都能提供读服务，唯一的区别是，Observer 机器不参与 Leader 选举过程，也不参与写操作的“过半写成功”策略，因此 Observer 可以在不影响写性能的情况下提升集群的读性能。

Leader 是整个 Zookeeper 集群工作机制的核心，主要工作有以下两个：
1）事务请求的唯一调度和处理者，保证集群事务处理的顺序性
2）集群内部各服务器的调度者
Follower 是 Zookeeper 集群状态的跟随者，其主要工作有以下三个：
1）处理客户端非事务请求，转发事务请求给 Leader
2）参与事务请求 Proposal 的投票
3）参与 Leader 选举

Observer 和 Follower 很像，唯一的区别是不参与任何形式的投票，只提供非事务服务通常用在不影响集群事务处理能力的前提下提升集群的非事务处理能力。

事务

在 Zookeeper 中，事务是指能够改变 Zookeeper 服务器状态的操作，我们称之为事务操作。对于每一个事务请求，Zookeeper 都会为其分配一个全局唯一的事务 iD，用 ZXID 表示，通常是一个 64 位的数字。每一个 ZXID 对应一次更新操作，从 ZXID 中可以识别出 Zookeeper 处理这些更新操作请求的全局顺序。

会话

Session 是指客户端会话。一个客户端连接是指客户端和服务器之间的一个 TCP 长连接。从第一次连接建立开始，客户端会话的生命周期也开始了，通过这个连接，客户端能够提供心跳检测与服务器保持有效的会话，也能够向 Zookeeper 服务器发送请求并接收响应，同时还能通过该连接接收来自服务器的 Watch 事件通知。Session 的 sessionTimeout 值用来设置一个客户端会话的超时时间。当由于服务器压力太大、网络故障或者客户端主动断开连接等各种原因导致客户端连接断开时，只要在 sessionTimeout 规定的时间内能够重新连接上集群中任意一台机器，那么之前创建的会话仍然有效。

数据节点 ZNode

Zookeeper 将所有数据存储在内存中，数据模型是一棵树，由/进行分割的路径，就是一个 ZNode（比如/foo/path1）。每个 ZNode 都会保存自己的数据内容，同时保存一系列属性信息。
ZNode 可以分为持久节点和临时节点。持久节点是指一旦这个 ZNode 被创建了，除非主动进行 ZNode 的移除操作，否则这个 ZNode 将一直保存在 Zookeeper 上。而临时节点的生命周期与客户端会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被移除。另外，Zookeeper 还允许用户为每个节点添加一个 SEQUENTIAL。一旦节点被标记上这个属性，那么在这个节点被创建的时候，Zookeeper 会自动在其节点名后面追加上一个整型数字，这个整型数字是一个由父节点维护的自增数字。
数据节点分为持久节点、临时节点和顺序节点，可以生成四种组合：持久、持久顺序、临时、临时顺序。

版本

对应每个 ZNode，Zookeeper 都会为其维护一个叫做 Stat 的数据结构，Stat 中记录了这个 ZNode 的三个数据版本，分别是 version（当前 ZNode 版本）、cversion（当前 ZNode 子节点的版本）和 aversion（当前 ZNode 的 ACL 版本）。还有 czxid（创建时的事务 ID），mzxid（最后一个被更新时的事务 ID）等。

version 为 0 表示自创建该节点后，被更新过 0 次。注意即使变更并没有使得数据内容的值发生变化，version 的值仍然会变更。
版本的作用可以用乐观锁原理来解释，更新数据的时候 ZK 使用乐观锁来保证原子性。

Watcher

Zookeeper 允许用户在指定节点上注册一些 Watcher，并且在一些特定事件触发的时候，Zookeeper 服务器会将事件通知到感兴趣的客户端上去。
Watcher 机制包括客户端线程、客户端 WatchManager 和 ZK 服务器三部分。客户端在向 ZK 服务器注册 Watcher 的同时，会将 Watcher 对象存储在客户端的 WatchManager 中。当 Zookeeper 服务器触发 Watcher 事件后，会向客户端发送通知，客户端线程从 WatchManager 中取出对应的 Watcher 对象来执行回调逻辑。
KeeperState 和 EventType 两个枚举分别代表了通知状态和事件类型。

表 7-3. Watcher 通知状态与事件类型一览

KeeperState	EventType	触发条件	说　明
SyncConnected (3)	None (-1)	客户端与服务器成功建立会话	此时客户端和服务器处于连接状态
	NodeCreated (1)	Watcher 监听的对应数据节点被创建	
	NodeDeleted (2)	Watcher 监听的对应数据节点被删除	
	NodeDataChanged (3)	Watcher 监听的对应数据节点的数据内容发生变更	
	NodeChildrenChanged (4)	Watcher 监听的对应数据节点的子节点列表发生变更	
Disconnected (0)	None (-1)	客户端与 ZooKeeper 服务器断开连接	此时客户端和服务器处于断开连接状态
Expired (-112)	None (-1)	会话超时	此时客户端会话失效，通常同时也会收到 SessionExpiredException 异常
AuthFailed (4)	None (-1)	通常有两种情况： ·使用错误的 scheme 进行权限检查。 ·SASL 权限检查失败。	通常同时也会收到 AuthFailedException 异常
~~Unknown~~ (-1)			从 3.1.0 版本开始已废弃
~~NoSyncConnected~~ (1)			

注意，客户端无法直接从该事件中获取到对应数据节点的原始数据内容以及变更后的新数据内容，而是需要客户端再次主动去重新获取数据。

Watcher 特性：
1）一次性：无论是服务器还是客户端，一旦一个 Watcher 被触发，ZK 都会将其从相应的存储中移除。因此开发人员必须要进行反复注册，这样的设计有效地减轻了服务器的压力
2）客户端串行执行：Watcher 的回调是一个串行同步的过程，这为我们保证了顺序。
3）轻量：WatchedEvent 是 ZK 整个 Watcher 通知机制的最小通知单元，只包含通知状态、

事件类型和节点路径。

ACL

Zookeeper 使用 ACL（访问控制列表）策略来进行权限控制，避免因误操作而带来的数据随意变更导致的分布式系统异常。
Zookeeper 定义了 5 种权限：
CREATE：创建节点
READ：读取节点数据和子节点列表
WRITE：更新节点数据的权限
DELETE：删除子节点
ADMIN：设置节点 ACL

ZAB 协议

ZAB：Zookeeper Atomic Broadcast 原子消息广播协议

核心

1） Zookeeper 使用一个单一的主进程来接收并处理客户端的所有事务请求，并采用 ZAB 的原子广播协议，将服务器数据的状态变更以事务 Proposal 的形式广播到所有的副本进程上去。ZAB 协议的这个主备模型架构保证了同一时刻集群中只能有一个主进程来广播服务器的状态变更，因此能够很好地处理客户端大量的并发请求。
2） 在分布式环境中，顺序执行的一些状态变更其前后存在一定的依赖关系，有些状态变更必须依赖于比它早生成的那些状态变更。ZAB 协议必须能够保证一个全局的变更序列被顺序地应用。
3）所有事务请求必须由一个全局唯一的服务器来协调处理，这样的服务器被称为 Leader 服务器，而余下的服务器则称为 Follower。Leader 负责将一个客户端事务请求转换成一个 Proposal，并将该 Proposal 分发给集群中所有的 Follower 服务器。之后 Leader 需要等待所有 Follower 的反馈，一旦超过半数的 Follower 进行了正确的反馈后，那么 Leader 就会再次向所有的 Follower 分发 Commit 消息，要求其将前一个 Proposal 进行提交。

内容

介绍

ZAB 协议包括两种基本的模式，分别是崩溃恢复和消息广播。当整个服务框架在启动过程中，或是当 Leader 出现网络中断、崩溃退出等异常情况时，ZAB 协议就会进入恢复模式，并选

举产生新的 Leader。当选举产生了新的 Leader 服务器，同时集群中已经有过半的机器与该 Leader 服务器完成了状态同步（数据同步）之后，ZAB 协议就会退出恢复模式。

当集群中已经有过半的 Follower 完成了与 Leader 的状态同步，那么整个服务器框架就可以进入消息广播模式。当一台同样遵循 ZAB 协议的服务器启动后加入到集群中时，如果此时集群中已经存在 Leader 在负责进行消息广播，那么新加入的服务器就会自觉进入数据恢复模式，找到 Leader，并与其进行数据同步，然后一起参与到消息广播流程中去。

Leader 在接收到客户端的事务请求后，会生成对应的 Proposal 并发起一轮广播协议；而如果集群中的其他机器接收到客户端的事务请求，那么这些非 Leader 服务器会首先将这个事务请求转发给 Leader。

当 Leader 崩溃或机器重启，或者集群中已经不存在过半的服务器与该 Leader 保持正常通信时，那么在重新开始新一轮的原子广播操作之前，所有进程首先会使用崩溃恢复协议来使彼此达到一个一致的状态，于是整个 ZAB 流程就会从消息广播模式进入到崩溃恢复模式。

消息广播

消息广播使用了一个原子广播协议，类似于 2PC。

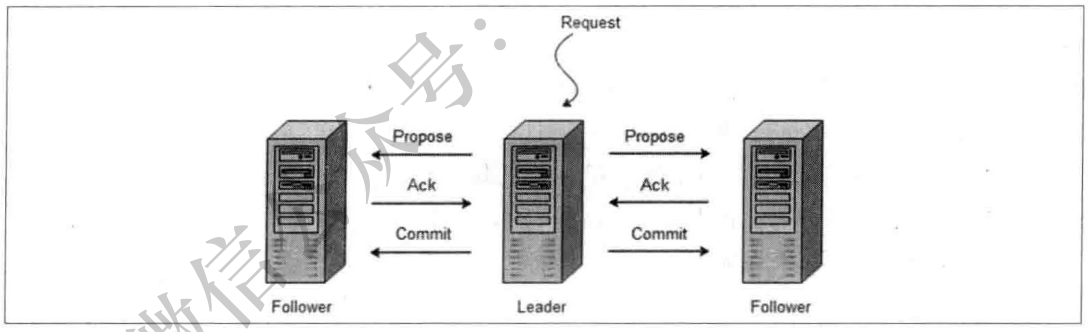


图 4-2. ZAB 协议消息广播流程示意图

针对客户端的事务请求，Leader 会为其生成对应的事务 Proposal，并将其发送给集群中其余所有的机器，然后再分别收集各自的选票，最后进行事务提交。
但是与 2PC 还是略有不同，移除了中断事务逻辑（rollback），所有的 Follower 要么正常反馈 Leader 提出的 Proposal，要么就抛弃 Leader；并且可以在过半的 Follower 已经反馈 ack 时就开始提交事务 Proposal 了，而不需要等待所有 Follower。
这样简化后的 2PC 是无法处理 Leader 崩溃而带来的数据不一致问题的，因此在 ZAB 协议中添加了崩溃恢复模式来解决这个问题。
另外，整个消息广播协议是基于具有 FIFO 特性的 TCP 协议来进行网络通信的，因此能够很容易地保证消息广播过程中消息接收与发送的顺序性。

在整个消息广播过程中，Leader 会为每个事务请求分配一个全局递增的唯一 ID，称为事务 ID（ZXID）。由于 ZAB 协议需要保证每一个消息严格的因果关系，因此必须将每个事务

Proposal 按照其 ZXID 的先后顺序来进行排序与处理。

在消息广播中，Leader 会为每个 Follower 都各自分配一个单独的队列，然后将需要广播的事务 Proposal 依次放入到这些队列中，并且按照 FIFO 策略进行消息发送。每个 Follower 在接收到这个事务 Proposal 之后，都会首先将其以事务日志的形式写入到本地磁盘中，并且在成功写入后反馈给 Leader 一个 ack。当 Leader 接收到过半数的 Follower 的 ack 响应后，就会广播一个 Commit 消息给所有的 Follower 以通知其进行事务提交，同时 Leader 自身也会完成对事务的提交，而每一个 Follower 在接收到 Commit 请求后，也会完成事务的提交。

崩溃恢复

ZAB 协议需要一个快速可靠的 Leader 选举算法，不仅需要让 Leader 自己知道自己被选举为 Leader，并且还需要让集群中的所有其他机器也能够快速感知到选举产生的 Leader 服务器。

1）ZAB 协议需要保证已经在 Leader 上提交的事务最终被所有服务器都提交

假设一个事务在 Leader 上被提交了，并且已经得到过半 Follower 的 ack，但是在它将 commit 消息发送给所有 Follower 之前，Leader 挂了。

2）ZAB 协议需要确保丢弃那些只在 Leader 上被提出的事务。
假设 Leader 提出了一个事务 Proposal 之后马上崩溃退出，从而导致集群中的其他服务器都没有收到这个事务 Proposal。于是，当 Leader 恢复后再次加入集群中的时候，ZAB 协议需要确保丢弃该事务。

基于以上两个特殊情况，需要保证 Leader 选举算法：能够确保提交**已经被 Leader 提交的事务** Proposal，同时丢弃已经被跳过的事务 Proposal。
如果让 Leader 选择算法能够保证新选举出来的 Leader 拥有集群中所有机器最高编号（ZXID 最大）的事务 Proposal，那么就可以保证这个新选举出来的 Leader 一定具有所有已经提交 Proposal，并且可以省去 Leader 服务器检查 Proposal 的提交和丢弃工作的这一步操作了。

Leader 选举

术语：SID 服务器 ID
ZXID 事务 ID，唯一标识一次服务器状态的变更
Vote 投票 当集群中的机器发现自己无法检测到 Leader 时，就会开始尝试进行投票
Quorum 过半机器数 如果总机器数为 n，那么 Quorum 为 n/2+1

进入 Leader 选举

当 Zookeeper 集群中的一台服务器出现以下两种情况之一时，就会开始进入 Leader 选举：
1）服务器 初始化
2）服务器运行时无法和 Leader 保持连接

而当一台机器进入 Leader 选举时，当前集群也可能处于以下两种状态：
1）集群中本来存在 Leader
2）集群中不存在 Leader

1）中通常是集群中的某一台机器启动比较晚，在它启动之前，集群已经可以正常工作，即已经存在了一台 Leader 服务器。针对这种情况，当该机器试图去选举 Leader 时，会被告知当前服务器的 Leader，对于该机器来说，仅仅需要和 Leader 机器建立起连接，并进行状态同步即可。

第一次投票

通常有两种情况会导致集群中不存在 Leader，一种情况是在整个服务器刚刚初始化情况时，此时尚未产生 Leader；另一种情况是 Leader 宕机。
此时，集群中的所有机器都处于一种试图选举出一个 Leader 的状态，这种状态称为 Looking。当一台机器处于 Looking 状态时，那么它就会向集群中的所有其他机器发送消息，这个消息称为投票。

投票可以表示为（SID，ZXID），SID 是所推举的服务器唯一 ID，ZXID 是事务 ID。
在第一次投票时，由于还无法检测到集群中其他机器的状态信息，因此每台机器都是将自己作为被推举的对象来进行投票。

第二次投票

集群中的每台机器发出自己的投票后，也会收到来自集群中其他机器的投票。每台机器都会根据一定的规则，来处理收到的其他机器的投票，并以此来决定是否需要变更自己的投票。
vote_sid：接收到的投票中所推举的 Leader 的 SID
vote_zxid：接收到的投票中所推举的 Leader 的 ZXID
self_sid：当前服务器自己的 SID
self_zxid：当前服务器自己的 ZXID

对于每次收到的投票的处理，都是一个对（vote_sid,vote_zxid）和（self_sid,self_zxid）的对比的过程。
规则 1：如果 vote_zxid 大于 self_zxid，那么认可当前收到的投票，并再次将该投票发送出去
规则 2：如果 vote_zxid 等于 self_zxid，则就对比两者的 SID。如果 vote_sid 大于 self_sid，那么就认可当前接收到的投票，并再次将该投票发送出去
规则 3：如果 vote_zxid 等于 self_zxid，并且 vote_sid 小于 self_sid，那么同样坚持自己的投票，不做变更。

确定 Leader

经过第二次投票后，集群中的每台机器都会再次受到其他机器的投票，然后开始统计投票。如果一台机器收到超过半数的相同的投票，那么整个投票对应的 SID 机器即为 Leader。

小结

哪台机器上的数据越新，那么越有可能成为 Leader。数据越新，ZXID 就越大，也就越能保证数据的恢复。如果几台服务器有着相同的 ZXID，那么 SID 较大的服务器成为 Leader。

数据同步

完成 Leader 选举之后，在接收客户端事务请求前，Leader 会确认事务日志中的所有 Proposal 是否已经被集群中过半的机器提交了，即是否完成数据同步。

正常情况下：Leader 会为每一个 Follower 都准备一个队列，并将那些没有被各 Follow 同步的事务以 Proposal 消息的形式逐个发送给 Follower，并在每一个 Proposal 消息后面紧接着再发送一个 Commit 消息，以表示该事务已经被提交。等到 Follower 将所有尚未同步的事务 Proposal 都从 Leader 上同步过来并成功应用到本地数据库中，Leader 就会将该 Follower 加入到真正的可用 Follower 列表中。

如何处理需要被丢弃的 Proposal 的？
ZXID 是一个 64 位的数字，低 32 位是一个单调递增的计数器，针对客户端的每一个事务请求，Leader 在产生一个新事务 Proposal 的时候，都会对该计数器进行加一操作；高 32 位则代表了 Leader 周期 epoch 编号，每当选举产生一个新的 Leader，就会从这个 Leader 上取出其本地日志中最大事务 Proposal 的 ZXID，并从该 ZXID 中解析出 epoch，然后再对其进行加一操作，之后就会以此编号作为新的 epoch，并将低 32 位置 0 来开始新的 ZXID。ZAB 协议中这一通过 epoch 编号来区分 Leader 周期变化的策略，能够避免不同的 Leader 错误地使用相同的 ZXID 编号提出不一样的事务 Proposal 的异常情况，这对于识别 Leader 崩溃恢复前后生成的 Proposal 非常有帮助。

当一个包含了上一个 Leader 周期中尚未提交过的事务 Proposal 的服务器启动时，其肯定无法成为 Leader，因为当前集群中一定包含一个 Quorum 集合（存在过半的处于 UP 状态的进程的进程子集），该集合中的机器一定包含了更高 epoch 的事务 Proposal，因此这台机器的事务 Proposal 一定不是最高，也就无法成为 Leader 了。当这台机器加入到集群后，以 Follower 的角色连接上 Leader 后，leader 会根据自己服务器上最后被提交的 Proposal 来与 Follower 的 Proposal 进行对比，对比的结果当然是 Leader 要求 Follower 进行一个回退操作——回退到一个已经被集群中过半机器提交的最新的事务 Proposal。

与 Paxos 算法的联系与区别

ZAB 协议并不是一个 Paxos 算法的典型实现。
联系：
1. 两者都存在一个类似 Leader 的角色，由其负责多个 Follower 的运行
2. Leader 都会等待过半 Follower 做出正确反馈后，才会将一个 Proposal 提交
3. 每个 Proposal 中都包含了一个表示当前 Leader 周期的值，ZAB 中称为 epoch，Paxos

中称为 ballot。

区别：
Paxos 算法中，一个新 Leader 会进行两个阶段的工作。第一阶段被称为读阶段，Leader 与所有其他机器进行通信来收集上一个 Leader 的 Proposal，并将它们提交；第二阶段被称为写阶段，Leader 开始提出自己的提案。
ZAB 协议中，在 Paxos 基础上增加了一个数据同步阶段。在同步阶段之前，ZAB 也存在一个类似于 Paxos 读阶段的过程，称为发现阶段。在同步阶段中，新的 Leader 会确保过半的 Follower 已经提交了之前 Leader 周期中的所有事务 Proposal，该阶段可以保证 Leader 在新的周期提出 Proposal 之前，所有机器都已经完成对之前所有事务 Proposal 的提交。一旦完成同步阶段后，那么 ZAB 就会执行和 Paxos 算法类似的写阶段。

本质区别是设计目标不一样，ZAB 主要用于构建一个高可用的分布式数据主备系统，Paxos 则用于构建一个分布式的一致性状态机系统。

典型应用

发布订阅（配置管理）

客户端向服务器注册自己需要关注的节点，一旦该节点的数据发生变更，那么客户端就会向响应的客户端发送 Watcher 时间通知，客户端接收到这个事件通知之后，需要主动到服务端获取最新的数据。
如果将配置信息存放到 Zookeeper 上进行集中管理，那么通常情况下，应用在启动的时候都会主动到 Zookeeper 服务端上进行一次配置信息的获取，同时，在指定节点上注册一个 Watcher 监听，这样的话，如果配置信息发生变更，服务端都会实时通知到所有订阅的客户端，从而达到获取最新配置信息的目的。

命名服务（软负载中心）

被命名的实体可以是集群中的机器、提供的服务地址或远程对象。

比如 RPC 的服务注册查找中心，ZK 客户端与 ZK 服务器保持连接，连接 ZK 服务器时 RPCServer 将自己的 IP 地址、端口号告诉 ZKServer，在 ZKServer 上创建节点。RPCClient 也连接 ZK 服务器，获取 RPCServer 的地址（软件负载均衡），并注册 Watcher。假如 RPCServer 挂掉，与 ZK 服务器的连接断开，那么临时节点会被删除，此时会通知 RPCClient，RPCClient 可以重新选择 RPCServer 进行连接。

并且 ZK 还可以创建全局唯一 ID，基于 ZK 提供的创建节点时使用 SEQUENTIAL 属性（顺序节点）。

集群管理

希望知道当前集群中有哪些集群在工作；
对集群中每天机器的运行时状态进行数据收集；
对集群中机器进行上下线通知

基于 ZK 提供的临时节点和 Watcher 监听特性，可以实现另一种集群机器存活性监控的系统。比如云主机管理，每台机器上线后，启动机器上的 Agent，向 ZK 的指定节点进行注册（临时节点），此时监控中心会接收到子节点变更事件，即上线通知，于是可以对这个新加入的机器开启相应的后台管理逻辑。另一方面，监控中心同样可以获取到机器下线的通知。

除了要对机器的在线状态进行检测，还要对机器的运行时状态进行监控。在运行的过程中，Agent 会定时将主机的运行状态写入 ZK 上的主机节点，监控中心提供订阅这些节点的数据变更通知来间接地获取主机的运行时信息。

Master 选举

在分布式系统中，Master 往往用来协调集群中的其他系统单元，具有对分布式系统状态变更的决定权。
场景：集群中的所有系统单元都需要对某个业务提供数据，但计算该数据的代价相当大，于是只希望一台机器或某几台机器来执行该任务，集群中的其他机器共享其计算结果。
那么怎么选择机器呢？Master 选举
一种方案是所有机器都向关系数据库插入某条数据，主键都是一样的，谁插入成功，谁就是 Master。但是无法感知 Master 挂掉的这种情形。
而 ZK 的强一致性可以保证在分布式高并发情况下节点的创建一定能够保证全局唯一性，即 ZK 将会保证客户端无法重复创建一个已经存在的数据节点(某个 path 下的节点名是唯一的)。只有一个客户端能够成功创建这个节点，那么这个客户端所在机器就成为了 Master。同时，其他没有在 ZK 上成功创建结点的客户端，都会在该节点的父节点上注册一个子节点变更的 Watcher，用于监控当前的 Master 机器是否存活，一旦发现当前的 Master 挂了，那么其余的客户端将会重新进行 Mater 选举。

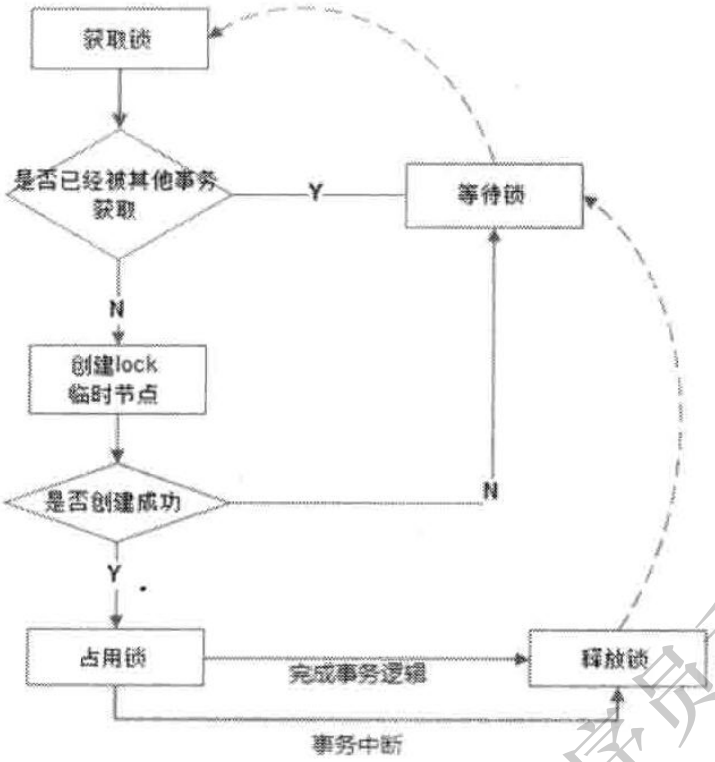
分布式锁

可以采用数据库的锁，只是数据库的性能很差。

排他锁

可以将一个数据节点（比如/exclusive_lock/lock）视为一个锁，所有客户端去创建该节点，创建成功的客户端就获得了这个锁。
如果获得锁的机器宕机，或者执行完业务逻辑后，都会将该节点删除。
无论什么情况下删除了该节点，都会通知所有在该节点（/exclusive_lock）上注册了监听子

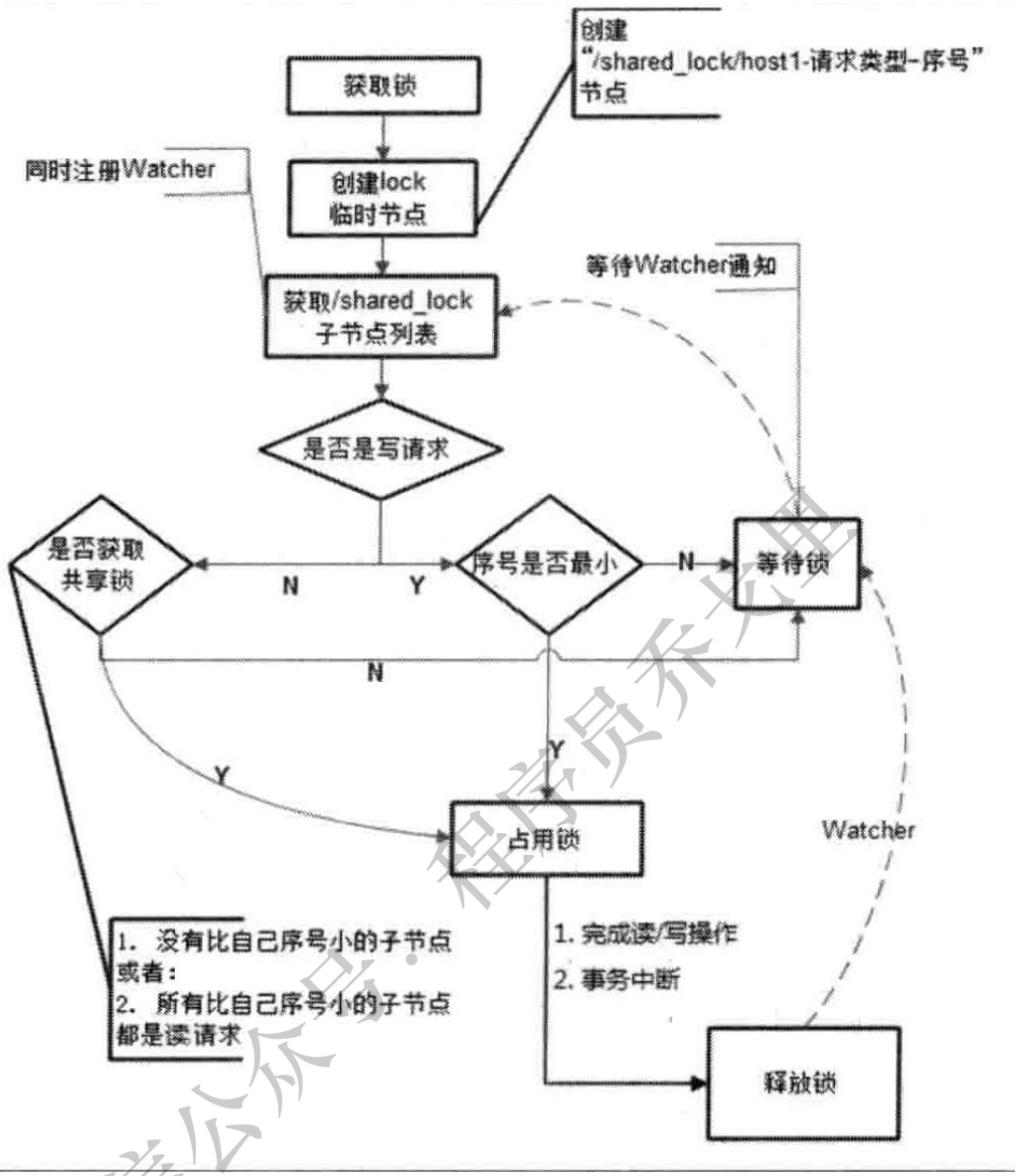
节点变更的 Watcher 的机器，这些机器在接收到通知后，会再次重新发起分布式锁的获取。



共享锁

同样是将数据节点视为一个锁，只是这里的数据节点是一个顺序节点，比如/shared_lock/{host_name}-请求类型-序列号。
获取锁时所有客户端都会到/shared_lock 这个节点下面创建一个顺序节点，读请求的话，请求类型为 R（写的话为 W）。
共享锁的定义是可以有读读，但不能有读写、写写、写读。
1、创建完节点后，获取/shared_lock 节点下的所有子节点，并对该节点注册了子节点变更的 Watcher 监听。
2、确定自己的节点序号在所有子节点中的顺序
3、对于读请求：如果没有比自己序号小的子节点，或者所有比自己序号小的节点都是读请求，那么表示自己已经获得了共享锁；如果比自己序号小的子节点中有写请求，那么就需要进入等待。
对于写请求：如果自己不是序号最小的子节点，那么就需要进入等待
4、接收到 Watcher 通知后，重复步骤 1

释放锁时就直接将自己的节点删除即可。



高可用

Zookeeper 集群服务器数推荐为奇数。由于过半存活即可用特性，3 台机器挂掉一台仍可用。而 5 台和 6 台服务器在挂掉两台后仍可用，挂掉三台都均不可用，因此 5 台和 6 台服务器在其容灾能力上是没有区别的，因此 Zookeeper 集群推荐部署为奇数台服务器。

Maven

maven 生命周期，常用 maven 命令；Maven 包冲突怎么解决？Maven 有什么命令可以解决包冲突？Maven 怎么把所有包打包

优缺点

Maven 有哪些优点和缺点
优点如下：

1. 简化了项目依赖管理：
2. 易于上手，对于新手可能一个"mvn clean package"命令就可能满足他的工作
3. 便于与持续集成工具（jenkins）整合
4. 便于项目升级，无论是项目本身升级还是项目使用的依赖升级。
5. 有助于多模块项目的开发，一个模块开发好后，发布到仓库，依赖该模块时可以直接从仓库更新，而不用自己去编译。
6. maven 有很多插件，便于功能扩展，比如生产站点，自动发布版本等

缺点如下：

1. maven 是一个庞大的构建系统，学习难度大
2. maven 采用约定优于配置的策略（convention over configuration），虽然上手容易，但是一旦出了问题，难于调试。

生命周期

Maven 有以下三种标准的生命周期：

- clean：主要用于清理上一次构建产生的文件，可以理解为删除 target 目录
  - pre-clean
  - clean
  - post-clean
- default(或 build)：
  - process-resources 默认处理 src/test/resources/下的文件，将其输出到测试的 classpath 目录中，
  - compile 编译 src/main/java 下的 java 文件，产生对应的 class，
  - process-test-resources 默认处理 src/test/resources/下的文件，将其输出到测试的 classpath 目录中，
  - test-compile 编译 src/test/java 下的 java 文件，产生对应的 class，
  - test 运行测试用例，
  - package 打包构件，即生成对应的 jar, war 等，
  - install 将构件部署到本地仓库，
  - deploy 部署构件到远程仓库
- site：用于创建新的文档，创建报告，部署网站
  - site 产生项目的站点文档
  - site-deploy 将项目的站点文档部署到服务器

目标代表一个特定的任务，它有助于项目的建设和管理。可以被绑定到零个或多个生成阶段。一个没有绑定到任何构建阶段的目标，它的构建生命周期可以直接调用执行。

Maven 坐标

一般 maven 使用[groupID,artifactId,version]来表示一个项目的某个版本

依赖范围（dependency-scope）

compile:编译依赖，默认的依赖方式，在**编译（编译项目和编译测试用例），运行测试用例，运行（项目实际运行）三个阶段都有效**，典型地有 spring-core 等 jar。
test:测试依赖，只在编译测试用例和运行测试用例有效，典型地有 JUnit。
provided:**对于编译和测试有效，不会打包进发布包中，典型的例子为 servlet-api**,一般的 web 工程运行时都使用容器的 servlet-api。
runtime:**只在运行测试用例和实际运行时有效，典型地是 jdbc 驱动 jar 包。**
system: **不从 maven 仓库获取该 jar,而是通过 systemPath 指定该 jar 的路径。**
import: 用于一个 dependencyManagement 对另一个 dependencyManagement 的继承。

多模块

配置一个打包类型为 pom 的聚合模块，然后在该 pom 中使用<module>元素声明要聚合的模块

多模块依赖管理

通过在父模块中声明 dependencyManagement 和 pluginManagement， 然后让子模块通过<parent>元素指定父模块，这样子模块在定义依赖是就可以只定义 groupId 和 artifactId，自动使用父模块的 version,这样统一整个项目的依赖的版本。

依赖冲突

一个项目的依赖来源于不同的组织，可能这些依赖还会依赖别的 Jar 包，如何保证这些传递依赖不会引起版本冲突。
使用<dependency>的<exclusion>元素将会引起冲突的元素排除。

SpringBoot

为什么要使用 SpringBoot

Spring Boot 是 Spring 旗下众多的子项目之一，其理念是约定优于配置，它通过实现了自动配置（大多数用户平时习惯设置的配置作为默认配置）的功能来为用户快速构建出标准化的应用。Spring Boot 的特点可以概述为如下几点：

1. 内置了嵌入式的 Tomcat、Jetty 等 Servlet 容器，应用可以不用打包成 War 格式，而是可以直接以 Jar 格式运行。
2. 提供了多个可选择的"starter"以简化 Maven 的依赖管理（也支持 Gradle），让您可以按需加载需要的功能模块。
3. 尽可能地进行自动配置，减少了用户需要动手写的各种冗余配置项，Spring Boot 提倡**无 XML 配置文件**的理念，使用 Spring Boot 生成的应用完全不会生成任何配置代码与 XML 配置文件。
4. 提供了一整套的对应用状态的监控与管理的功能模块（通过引入 spring-boot-starter-actuator），包括应用的线程信息、内存信息、应用是否处于健康状态等，为了满足更多的资源监控需求，Spring Cloud 中的很多模块还对其进行了扩展。

@Conditional

@Conditional 是由 Spring 4 提供的一个新特性，用于根据特定条件来控制 Bean 的创建行为。而在我们开发基于 Spring 的应用的时候，难免会需要根据条件来注册 Bean。
使用到@Conditional 注解来提供更加灵活的条件判断，例如以下几个判断条件：

- 在类路径中是否存在这样的一个类。
- 在 Spring 容器中是否已经注册了某种类型的 Bean（如未注册，我们可以让其自动注册到容器中，上一条同理）。
- 一个文件是否在特定的位置上。
- 一个特定的系统属性是否存在。
- 在 Spring 的配置文件中是否设置了某个特定的值。

而且可以自定义 Condition，只需要去实现 Condition 接口即可。
@SpringBootApplication 是一个组合注解，@Configuration、@EnableAutoConfiguration 与 @ComponentScan 三个注解（如果我们想定制自定义的自动配置实现，声明这三个注解就足够了），而@EnableAutoConfiguration 是我们的关注点，从它的名字可以看出来，它是用来开启自动配置的。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {

   String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

   /**
    * Exclude specific auto-configuration classes such that they will never be
applied.
    * @return the classes to exclude
```

```
    */
   Class<?>[] exclude() default {};

   /**
    * Exclude specific auto-configuration class names such that they will never
be
    * applied.
    * @return the class names to exclude
    * @since 1.3.0
    */
   String[] excludeName() default {};

}
```

@Import（Spring 提供的一个注解，可以导入配置类或者 Bean 到当前类中）导入了 EnableAutoConfigurationImportSelector 类，根据名字来看，它应该就是我们要找到的目标了。不过查看它的源码发现它已经被 Deprecated 了，而官方 API 中告知我们去查看它的父类 **AutoConfigurationImportSelector#selectImports**。

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
   if (!isEnabled(annotationMetadata)) {
      return NO_IMPORTS;
   }
   try {
      AutoConfigurationMetadata autoConfigurationMetadata =
AutoConfigurationMetadataLoader
            .loadMetadata(this.beanClassLoader);
      AnnotationAttributes attributes = getAttributes(annotationMetadata);
      List<String> configurations =
getCandidateConfigurations(annotationMetadata,
            attributes);
      configurations = removeDuplicates(configurations);
      configurations = sort(configurations, autoConfigurationMetadata);
      Set<String> exclusions = getExclusions(annotationMetadata, attributes);
      checkExcludedClasses(configurations, exclusions);
      configurations.removeAll(exclusions);
      configurations = filter(configurations, autoConfigurationMetadata);
      fireAutoConfigurationImportEvents(configurations, exclusions);
      return configurations.toArray(new String[configurations.size()]);
   }
   catch (IOException ex) {
      throw new IllegalStateException(ex);
   }
}
```

重点在于方法 getCandidateConfigurations()返回了自动配置类的信息列表，而它通过调用**SpringFactoriesLoader.loadFactoryNames()**来扫描加载含有 META-INF/spring.factories 文件的 jar 包，该文件记录了具有哪些自动配置类。

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
      AnnotationAttributes attributes) {
   List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
         getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
   Assert.notEmpty(configurations,
         "No auto configuration classes found in META-INF/spring.factories. If
you "
               + "are using a custom packaging, make sure that file is correct.");
   return configurations;
}
```

```
public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader
classLoader) {
   String factoryClassName = factoryClass.getName();
   try {
      Enumeration<URL> urls = (classLoader != null ?
classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
            ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
      List<String> result = new ArrayList<String>();
      while (urls.hasMoreElements()) {
         URL url = urls.nextElement();
         Properties properties = PropertiesLoaderUtils.loadProperties(new
UrlResource(url));
         String factoryClassNames = properties.getProperty(factoryClassName);

result.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray(fac
toryClassNames)));
      }
      return result;
   }
   catch (IOException ex) {
      throw new IllegalArgumentException("Unable to load [" +
factoryClass.getName() +
            "] factories from location [" + FACTORIES_RESOURCE_LOCATION + "]",
ex);
   }
}
```

JWT

传统身份验证的方法

HTTP 是一种没有状态的协议，也就是它并不知道是谁是访问应用。这里我们把用户看成是客户端，客户端使用用户名还有密码通过了身份验证，不过下回这个客户端再发送请求时候，还得再验证一下。

解决的方法就是，当用户请求登录的时候，如果没有问题，我们在服务端生成一条记录，这个记录里可以说明一下登录的用户是谁，然后把这条记录的 ID 号发送给客户端，客户端收到以后把这个 ID 号存储在 Cookie 里，下次这个用户再向服务端发送请求的时候，可以带着这个 Cookie ，这样服务端会验证一个这个 Cookie 里的信息，看看能不能在服务端这里找到对应的记录，如果可以，说明用户已经通过了身份验证，就把用户请求的数据返回给客户端。

上面说的就是 Session，我们需要在服务端存储为登录的用户生成的 Session ，这些 Session 可能会存储在内存，磁盘，或者数据库里。我们可能需要在服务端定期的去清理过期的 Session 。
前端退出的话就清 cookie。后端强制前端重新认证的话就清或者修改 session。

如果是分布式部署，需要做多机共享 session 机制，实现方法可将 session 存储到数据库中或者 redis 中
基于 cookie 的机制很容易被 CSRF

存储

session、cookie、sessionStorage、localstorage 的区别
session: 主要存放在服务器端，相对安全
cookie: 可设置有效时间，默认是关闭浏览器后失效，主要存放在客户端，并且不是很安全，可存储大小约为 4kb
sessionStorage: 仅在当前会话下有效，关闭页面或浏览器后被清除
localstorage: 除非被清除，否则永久保存

基于 Token 的身份验证方法

使用基于 Token 的身份验证方法，在服务端不需要存储用户的登录记录。大概的流程是这样的：

1. 客户端使用用户名跟密码请求登录
2. 服务端收到请求，去验证用户名与密码
3. 验证成功后，服务端会签发一个 Token，再把这个 Token 发送给客户端

4. 客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者 Local Storage 里
5. 客户端每次向服务端请求资源的时候需要带着服务端签发的 Token，放入 HTTP Header 中的 Authorization 位
6. 服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据

JWT，读作：jot ，表示：JSON Web Tokens 。JWT 标准的 Token 有三个部分：

header
payload
signature

中间用点分隔开，并且都会使用 Base64 编码，所以真正的 Token 看起来像这样：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJuaW5naGFvLm5ldCIsImV4cCI6IjE0Mzg5NTU0NDU
iLCJuYW1lIjoid2FuZ2hhbyIsImFkbWluIjp0cnVlfQ.SwyHTEx_RQppr97g4J5lKXtabJecpejuef8AqKYMAJc
```

标准

Header

header 部分主要是两部分内容，一个是 Token 的类型，另一个是使用的算法，比如下面类型就是 JWT，使用的算法是 HS256。

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

Payload（有效载荷）

Payload 里面是 Token 的具体内容，这些内容里面有一些是标准字段，你也可以添加其它需要的内容。下面是标准字段：

iss：Issuer，发行者
sub：Subject，主题
aud：Audience，观众
exp：Expiration time，过期时间
nbf：Not before
iat：Issued at，发行时间
jti：JWT ID

Signature

Signature 部分其实就是对我们前面的 Header 和 Payload 部分进行签名，保证 Token 在传输的过程中没有被篡改或者损坏，签名的算法也很简单，但是，为了加密，所以除了 Header 和 Payload 之外，还多了一个密钥字段，完整算法为：

```
Signature = HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
    secret)
```

这部分内容有三个部分，先是用 Base64 编码的 **header.payload** ，再用加密算法加密一下，加密的时候要放进去一个 Secret 密钥，这个密钥被存储在服务端。

使用注意

1. payload 中不要存放敏感信息，因为可以被很容易地获取到；
2. Token 在服务器没有存储，只能通过设置过期时间来让 Token 失效，也就是没有办法让 Token 主动失效，就没有办法做单点登录（可以考虑服务器进行存储）
3. 保护好 secret 私钥，该私钥非常重要。
4. 如果可以，请使用 HTTPS 协议，不！是务必使用 HTTPS!

比较

可扩展性

随着应用程序的扩大和用户数量的增加，你必将开始水平或垂直扩展。session 数据通过文件或数据库存储在服务器的内存中。在水平扩展方案中，你必须开始复制服务器数据，你必须创建一个独立的中央 session 存储系统，以便所有应用程序服务器都可以访问。否则，由于 session 存储的缺陷，你将无法扩展应用程序。解决这个挑战的另一种方法是使用 sticky session。你还可以将 session 存储在磁盘上，使你的应用程序在云环境中轻松扩展。这类解决方法在现代大型应用中并没有真正发挥作用。建立和维护这种分布式系统涉及到深层次的技术知识，并随之产生更高的财务成本。在这种情况下，使用 JWT 是无缝的;由于基于 token 的身份验证是无状态的，所以不需要在 session 中存储用户信息。我们的应用程序可以轻松扩展，因为我们可以使用 token 从不同的服务器访问资源，而不用担心用户是否真的登录到某台服务器上。**你也可以节省成本，因为你不需要专门的服务器来存储 session。为什么？因为没有 session！**

注意：如果你正在构建一个小型应用程序，这个程序完全不需要在多台服务器上扩展，并且不需要 RESTful API 的，那么 session 机制是很棒的。 如果你使用专用服务器运行像 Redis 那样的工具来存储 session，那么 session 也可能会为你完美地运作！

安全性

JWT 签名旨在防止在客户端被篡改，但也可以对其进行加密，以确保 token 携带的 claim 非常安全。JWT 主要是直接存储在 web 存储（本地/session 存储）或 cookies 中。 JavaScript 可以访问同一个域上的 Web 存储。这意味着你的 JWT 可能容易受到 XSS（跨站脚本）攻击。恶意 JavaScript 嵌入在页面上，以读取和破坏 Web 存储的内容。事实上，很多人主张，由于 XSS 攻击，一些非常敏感的数据不应该存放在 Web 存储中。一个非常典型的例子是确保你的 JWT 不将过于敏感/可信的数据进行编码，例如用户的社会安全号码。

最初，我提到 JWT 可以存储在 cookie 中。事实上，JWT 在许多情况下被存储为 cookie，并且 **cookies 很容易受到 CSRF（跨站请求伪造）攻击**。预防 CSRF 攻击的许多方法之一是确保你的 cookie 只能由你的域访问。作为开发人员，**不管是否使用 JWT，确保必要的 CSRF 保护措施到位以避免这些攻击。**

现在，JWT 和 session ID 也会暴露于未经防范的重放攻击。建立适合系统的重放防范技术，完全取决于开发者。解决这个问题的一个方法是确保 JWT 具有短期过期时间。虽然这种技术并不能完全解决问题。然而，解决这个挑战的其他替代方案是将 JWT 发布到特定的 IP 地址并使用浏览器指纹。

注意：使用 HTTPS / SSL 确保你的 Cookie 和 JWT 在客户端和服务器传输期间默认加密。这有助于避免中间人攻击！

RESTful API 服务

现代应用程序的常见模式是从 RESTful API 查询使用 JSON 数据。目前大多数应用程序都有 RESTful API 供其他开发人员或应用程序使用。由 API 提供的数据具有几个明显的优点，其中之一就是这些数据可以被多个应用程序使用。在这种情况下，传统的使用 session 和 Cookie 的方法在用户认证方面效果不佳，因为它们将状态引入到应用程序中。

RESTful API 的原则之一是它应该是无状态的，这意味着当发出请求时，总会返回带有参数的响应，不会产生附加影响。用户的认证状态引入这种附加影响，这破坏了这一原则。保持 API 无状态，不产生附加影响，意味着维护和调试变得更加容易。

另一个挑战是，由一个服务器提供 API，而实际应用程序从另一个服务器调用它的模式是很常见的。**为了实现这一点，我们需要启用跨域资源共享（CORS）。Cookie 只能用于其发起的域，相对于应用程序，对不同域的 API 来说，帮助不大。在这种情况下使用 JWT 进行身份验证可以确保 RESTful API 是无状态的**，你也不用担心 API 或应用程序由谁提供服务。

性能

当从客户端向服务器发出请求时，如果大量数据在 JWT 内进行编码，则每个 HTTP 请求都会产生大量的开销。编码时，JWT 的大小将是 SESSION ID（标识符）的几倍，从而在每个 HTTP 请求中，JWT 比 SESSION ID 增加更多的开销。

实效性

JWT 是一种无状态身份验证机制，因为用户状态永远不会保存在服务器内存中。 由于 JWT 是独立的，所有必要的信息都在那里，所以减少了多次查询数据库的需求。

此外，无状态 JWT 的实效性相比 session 太差，只有等到过期才可销毁，而 session 则可手动销毁。
例如有个这种场景，如果 JWT 中存储有权限相关信息，比如当前角色为 admin，但是由于 JWT 所有者滥用自身权利，高级管理员将权利滥用者的角色降为 user。但是由于 JWT 无法实时刷新，必需要等到 JWT 过期，强制重新登录时，高级管理员的设置才能生效。
或者是用户发现账号被异地登录，然后修改密码，此时 token 还未过期，异地的账号一样可以进行操作包括修改密码。
但这种场景也不是没有办法解决，解决办法就是将 JWT 生成的 token 存入到 redis 或者数据库中，当用户登出或作出其他想要让 token 失效的举动，可通过删除 token 在数据库或者 redis 里面的对应关系来解决这个问题。

Lombok

Lombok 这个东西工作却在编译期，在运行时是无法通过反射获取到这个注解的。
而且由于他相当于是在编译期对代码进行了修改，因此从直观上看，源代码甚至是语法有问题的。
Lombok 的基本流程应该基本是这样：

1. 定义编译期的注解
2. 利用 JSR269 api(Pluggable Annotation Processing API)创建编译期的注解处理器
3. 利用 tools.jar 的 javac api 处理 AST(抽象语法树)
4. 将功能注册进 jar 包

开发步骤（编译器修改 AST）

1) 定义注解

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.SOURCE)
public @interface Getter {
}
```

编译期的注解！

2) 定义注解处理器

```
@SupportedAnnotationTypes("com.mythsman.test.Getter")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class GetterProcessor extends AbstractProcessor {
    @Override
    public synchronized void init(ProcessingEnvironment
processingEnv) {
super.init(processingEnv);
    this.messager = processingEnv.getMessager();
    this.trees = JavacTrees.instance(processingEnv);
    Context context = ((JavacProcessingEnvironment)
processingEnv).getContext();
    this.treeMaker = TreeMaker.instance(context);
    this.names = Names.instance(context);

    }
    @Override
```

```
    public boolean process(Set<? extends TypeElement> annotations,
RoundEnvironment roundEnv) {
        Set<? extends Element> set =
roundEnv.getElementsAnnotatedWith(Getter.class);
    set.forEach(element -> {
        JCTree jcTree = trees.getTree(element);
        jcTree.accept(new TreeTranslator() {
            @Override
            public void visitClassDef(JCTree.JCClassDecl
jcClassDecl) {
                List<JCTree.JCVariableDecl> jcVariableDeclList =
List.nil();
                for (JCTree tree : jcClassDecl.defs) {
                    if (tree.getKind().equals(Tree.Kind.VARIABLE))
{
                        JCTree.JCVariableDecl jcVariableDecl =
(JCTree.JCVariableDecl) tree;
                        jcVariableDeclList =
jcVariableDeclList.append(jcVariableDecl);
                    }
                }
                jcVariableDeclList.forEach(jcVariableDecl -> {
                    messager.printMessage(Diagnostic.Kind.NOTE,
jcVariableDecl.getName() + " has been processed");
                    jcClassDecl.defs =
jcClassDecl.defs.prepend(makeGetterMethodDecl(jcVariableDecl))
;
                });
                super.visitClassDef(jcClassDecl);
            }
        });
    });

        return true;
    }

    private JCTree.JCMethodDecl
makeGetterMethodDecl(JCTree.JCVariableDecl jcVariableDecl) {
    ListBuffer<JCTree.JCStatement> statements = new
ListBuffer<>();

statements.append(treeMaker.Return(treeMaker.Select(treeMaker.
Ident(names.fromString("this")), jcVariableDecl.getName())));
```

```
    JCTree.JCBlock body = treeMaker.Block(0,
statements.toList());
    return
treeMaker.MethodDef(treeMaker.Modifiers(Flags.PUBLIC),
getNewMethodName(jcVariableDecl.getName()),
jcVariableDecl.vartype, List.nil(), List.nil(), List.nil(), body
null);
}

    private Name getNewMethodName(Name name) {
    String s = name.toString();
    return names.fromString("get" + s.substring(0,
1).toUpperCase() + s.substring(1, name.length()));
}

}
```

需要定义两个注解，一个表示该处理器需要处理的注解，另外一个表示该处理器支持的源码版本。然后需要着重实现两个方法，init 跟 process。init 的主要用途是通过 ProcessingEnvironment 来获取编译阶段的一些环境信息;process 主要是实现具体逻辑的地方，也就是对 AST 进行处理的地方。

init：
Messager 主要是用来在编译期打 log 用的
JavacTrees 提供了待处理的抽象语法树
TreeMaker 封装了创建 AST 节点的一些方法
Names 提供了创建标识符的方法

Process：

1. 利用 roundEnv 的 getElementsAnnotatedWith 方法过滤出被 Getter 这个注解标记的类，并存入 set
2. 遍历这个 set 里的每一个元素，并**生成 jCTree 这个语法树**
3. 创建一个 TreeTranslator，并重写其中的 visitClassDef 方法，这个方法处理遍历语法树得到的**类定义部分 jcClassDecl**
4. 创建一个 **jcVariableDeclList 保存类的成员变量**
5. 遍历 jcTree 的所有成员(包括成员变量和成员函数和构造函数)，**过滤出其中的成员变量，并添加进 jcVariableDeclList**
6. 将 jcVariableDeclList 的所有变量**转换成需要添加的 getter 方法，并添加进 jcClassDecl 的成员中**
7. 调用默认的遍历方法遍历处理后的 jcClassDecl
8. 利用上面的 TreeTranslator 去处理 jcTree

3) 执行

Getter.java 是注解类没问题，但是 GetterProcessor.java 是处理器，App.java 需要在编译期调用这个处理器，因此这两个东西是不能一起编译的，正确的编译方法应该是类似下面这样，写成 compile.sh 脚本就是：

```
mkdir classes
javac -cp $JAVA_HOME/lib/tools.jar com/mythsman/test/Getter* -d
classes/
javac -cp classes -d classes -processor
com.mythsman.test.GetterProcessor com/mythsman/test/App.java
javap -p classes com/mythsman/test/App.class
java -cp classes com.mythsman.test.App
```

1. 创建保存 class 文件的文件夹
2. 导入 tools.jar，编译 processor 并输出
3. 编译 App.java，并使用 javac 的-processor 参数指定编译阶段的处理器 GetterProcessor
4. 用 javap 显示编译后的 App.class 文件(非必须，方便看结果)
5. 执行测试类

4）构建

这应当是两个项目，一个是 processor 项目，这个项目应当被打成一个 jar 包，供调用者使用；另一个项目是 app 项目，这个项目是专门使用 jar 包的，他并不希望添加任何额外编译参数，就跟 lombok 的用法一样。
简单来说，就是我们希望把 processor 打成一个包，并且在使用时不需要添加额外参数。
那么如何在调用的时候不用加参数呢，其实我们知道 java 在编译的时候会去资源文件夹下读一个 META-INF 文件夹，这个文件夹下面除了 MANIFEST.MF 文件之外，还可以添加一个 services 文件夹，我们可以在这个文件夹下创建一个文件，文件名是 javax.annotation.processing.Processor，文件内容是 com.mythsman.test.GetterProcessor。
我们知道 maven 在编译前会先拷贝资源文件夹，然后当他在编译时候发现了资源文件夹下的 META-INF/serivces 文件夹时，他就会读取里面的文件，并将文件名所代表的接口用文件内容表示的类来实现。这就相当于做了-processor 参数该做的事了。

```
.
├── pom.xml
├── src
│   └── main
│       ├── java
│       │   └── com
│       │       └── mythsman
│       │           └── test
│       │               ├── Getter.java
│       │               └── GetterProcessor.java
│       └── resources
```

```
|          └── META-INF
|              └── services
|                  └── javax.annotation.processing.Processor
```

```
<resources>
       <resource>
          <directory>src/main/resources</directory>
          <excludes>
             <exclude>META-INF/**/*</exclude>
          </excludes>
       </resource>
    </resources>
    <plugins>
       <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-resources-plugin</artifactId>
          <version>2.6</version>
          <executions>
             <execution>
                <id>process-META</id>
                <phase>prepare-package</phase>
                <goals>
                   <goal>copy-resources</goal>
                </goals>
                <configuration>

<outputDirectory>target/classes</outputDirectory>
                   <resources>
                      <resource>

<directory>${basedir}/src/main/resources/</directory>
                         <includes>
                            <include>**/*</include>
                         </includes>
                      </resource>
                   </resources>
                </configuration>
             </execution>
          </executions>
       </plugin>
       ...
    </plugins>
</build>
```

maven 构建的第一步就是调用 maven-resources-plugin 插件的 resources 命令，将 resources 文件夹复制到 target/classes 中，那么我们配置一下 resources 标签，过滤掉 META-INF 文件夹，这样在**编译的时候就不会找到 services 的配置了**。然后我们在**打包前(prepare-package 生命周期)再利用 maven-resources-plugin 插件的 copy-resources 命令把 services 文件夹重新拷贝过来不就好了么。**