在《分布式服务化系统一致性的“最佳实干”》一文中提出了保证系统最终一致性的定期校对模式，在定期校对模式中最常使用的方法是在每个系统间传递和保存一个统一的唯一流水号（或称为traceid），通过系统间两两核对或者第三方统一核对唯一流水号来保证各个系统之间步伐一致、没有掉队的行为，也就是系统间状态一致，在互联网的世界里，产生唯一流水号的服务系统俗称发号器。Twitter的Snowflake是一个流行的开源的发号器的实现。Slowfake是由Scala语言实现的，并且文档简单、发布模式单一、缺少支持和维护，很难在现实的项目中直接使用。

为了能让Java领域的小伙伴们在不同的环境下快速使用发号器服务，本文向大家推荐一款自主研发的多场景分布式发号器Vesta，这是由Java语言编写的，可以通过Jar包的形式嵌入到任何Java开发的项目中，也可以通过服务化或者REST服务发布，发布样式灵活多样，使用简单、方便、高效。

Vesta是一款通用的唯一流水号产生器，它具有全局唯一、粗略有序、可反解和可制造等特性，它支持三种发布模式：嵌入发布模式、中心服务器发布模式、REST发布模式，根据业务的性能需求，它可以产生最大峰值型和最小粒度型两种类型的ID，它的实现架构使其具有高性能，高可用和可伸缩等互联网产品需要的质量属性，是一款通用的高性能的发号器产品。

本文聚焦在笔者原创的多场景分布式发号器Vesta的设计、实现、性能评估等方面，同时介绍Vesta的发布模式以及使用方式，并在最后给读者介绍如何在你的项目中使用Vesta。

为什么不用UUID

UUID虽然能够保证ID的唯一性，但是，它无法满足业务系统需要的很多其他特性，例如：时间粗略有序性，可反解和可制造型。另外，UUID产生的时候使用完全的时间数据，性能比较差，并且UUID比较长，占用空间大，间接导致数据库性能下降，更重要的是，UUID并不具有有序性，这导致B+树索引在写的时候会有过多的随机写操作（连续的ID会产生部分顺序写），另外写的时候由于不能产生顺序的append操作，需要进行insert操作，这会读取整个B+树节点到内存，然后插入这条记录后写整个节点回磁盘，这种操作在记录占用空间比较大的情况下，性能下降比较大，具体压测报告请参考：Mysql性能压测实践报告

既然数据库自增ID和UUID有诸多的限制，我们需要整理一下发号器的需求。

需求分析和整理

既然数据库自增ID和UUID有诸多的限制，我们需要整理一下发号器的需求。

需求是所有设计的起始点，一切偏离需求的设计，都是“耍流氓”，下面是我们总结的分布式微服务系统里面对发号器的需求：

全局唯一

有些业务系统可以使用相对小范围的唯一性，例如，如果用户是唯一的，那么同一用户的订单采用自增序列在用户范围内也是唯一的，但是如果这样设计，订单系统就会在逻辑上依赖用户系统，因此，不如我们保证ID在系统范围内的全局唯一性更实用。

分布式系统保证全局唯一的一个悲观策略是使用锁或者分布式锁，但是，只要使用了锁，就会大大的降低性能。

因此，我们决定利用时间的有序性，并且在时间的某个单元下采用自增序列，达到全局的唯一性。

粗略有序

上面讨论了UUID的最大问题就是无序的，任何业务都希望生成的ID是有序的，但是，分布式系统中要做到完全有序，就涉及到数据的汇聚，当然要用到锁或者布式锁，考虑到效率，只能采用折中的方案，粗略有序，到底有多粗略，目前有两种主流的方案，一种是秒级有序，一种是毫秒级有序，这里又有一个权衡和取舍，我们决定支持两种方式，通过配置来决定服务使用其中的一种方式。

可反解

一个 ID 生成之后，ID本身带有很多信息量，线上排查的时候，我们通常首先看到的是ID，如果根据ID就能知道什么时候产生的，从哪里来的，这样一个可反解的 ID 可以帮上很多忙。

如果ID 里有了时间而且能反解，在存储层面就会省下很多传统的timestamp 一类的字段所占用的空间了，这也是一举两得的设计。

可制造

一个系统即使再高可用也不会保证永远不出问题，出了问题怎么办，手工处理，数据被污染怎么办，洗数据，可是手工处理或者洗数据的时候，假如使用数据库自增字段，ID已经被后来的业务覆盖了，怎么恢复到系统出问题的时间窗口呢？

所以，我们使用的发号器一定要可复制，可恢复 ，可制造。

高性能

不管哪个业务，订单也好，商品也好，如果有新记录插入，那一定是业务的核心功能，对性能的要求非常高，ID生成取决于网络IO和CPU的性能，CPU一般不是瓶颈，根据经验，单台机器TPS应该达到10000/s。

高可用

首先，发号器必须是一个对等的集群，一台机器挂掉，请求必须能够转发到其他机器，另外，重试机制也是必不可少的。最后，如果远程服务宕机，我们需要有本地的容错方案，本地库的依赖方式可以作为高可用的最后一道屏障。

可伸缩

作为一个分布式系统，永远都不能忽略的就是业务在不断地增长，业务的绝对容量不是衡量一个系统的唯一标准，要知道业务是永远增长的，所以，系统设计不但要考虑能承受的绝对容量，还必须考虑业务增长的速度，系统的水平伸缩是否能满足业务的增长速度是衡量一个系统的另一个重要标准。

架构原理与设计

本节聚焦在如何亲自设计和实现一款高效的多场景分布式发号器，从对分布式多场景发号器的需求整理开始，挖掘互联网企业对分布式发号器的期待和需求，并根据确定的核心需求和特色需求，提出设计的解决方案，其中，设计考虑到了不同的用户、不同的性能场景、不同的配置模式和不同的环境的差异，力求设计一款多功能、多场景的高性能的互联网发号器，以Java语言为基础，给出一个发号器的参考实现，让Java领域的小伙伴们在不同的环境下可以快速的使用和集成发号器服务，发号器的参考实现作为一个通用的开源项目，对不同的使用方式提供了相应的用户向导。

发布模式

根据最终的客户使用方式，可分为嵌入发布模式，中心服务器发布模式和REST发布模式。

> 1. **嵌入发布模式**：只适用于Java客户端，提供一个本地的Jar包，Jar包是嵌入式的原生服务，需要提前配置本地机器ID（或者服务启动时候Zookeeper动态分配唯一的ID,在第二版中实现），但是不依赖于中心服务器。
> 2. **中心服务器发布模式**：只适用于Java客户端，提供一个服务的客户端Jar包，Java程序像调用本地API一样来调用，但是依赖于中心的ID产生服务器。
> 3. **REST发布模式**：中心服务器通过Restful API导出服务，供非Java语言客户端使用。

发布模式最后会记录在生成的ID中。也参考下面数据结构段的发布模式相关细节。

ID类型

根据时间的位数和序列号的位数，可分为最大峰值型和最小粒度型。

最大峰值型：采用秒级有序，秒级时间占用30位，序列号占用20位。

字段	版本	类型	生成方式	秒级时间	序列号	机器ID

位数	*63*	*62*	*60-61*	*40-59*	*10-39*	*0-9*

最小粒度型：采用毫秒级有序，毫秒级时间占用40位，序列号占用10位。

字段	版本	类型	生成方式	毫秒级时间	序列号	机器**ID**
位数	*63*	*62*	*60-61*	*20-59*	*10-19*	*0-9*

最大峰值型能够承受更大的峰值压力，但是粗略有序的粒度有点大，最小粒度型有较细致的粒度，但是每个毫秒能承受的理论峰值有限，为1k，同一个毫秒如果有更多的请求产生，必须等到下一个毫秒再响应。

ID类型在配置时指定，需要重启服务才能互相切换。

数据结构

机器ID

10位， 2^10=1024, 也就是最多支持1000+个服务器。中心发布模式和REST发布模式一般不会有太多数量的机器，按照设计每台机器TPS 1万/s，10台服务器就可以有10万/s的TPS，基本可以满足大部分的业务需求。

但是考虑到我们在业务服务可以使用内嵌发布方式，对机器ID的需求量变得更大，这里最多支持1024个服务器。

序列号

最大峰值型：

20位，理论上每秒内平均可产生2^20= 1048576个ID，百万级别，如果系统的网络IO和CPU足够强大，可承受的峰值达到每毫秒百万级别。

最小粒度型：

10位，每毫秒内序列号总计2^10=1024个, 也就是每个毫秒最多产生1000+个ID，理论上承受的峰值完全不如我们最大峰值方案。

秒级时间/毫秒级时间

最大峰值型：

30位，表示秒级时间，2^30/60/60/24/365=34，也就是可使用30+年。

最小粒度型：

40位，表示毫秒级时间，2^40/1000/60/60/24/365=34，同样可以使用30+年。

生成方式

2位，用来区分三种发布模式：嵌入发布模式，中心服务器发布模式，REST发布模式。

> **00**：嵌入发布模式
>
> **01**：中心服务器发布模式
>
> **02**：REST发布模式
>
> **03**：保留未用

ID类型

1位，用来区分两种ID类型：最大峰值型和最小粒度型。

> **0**：最大峰值型
>
> **1**：最小粒度型

版本

1位，用来做扩展位或者扩容时候的临时方案。

> **0**：默认值，以免转化为整型再转化回字符串被截断
>
> **1**：表示扩展或者扩容中

作为30年后扩展使用，或者在30年后ID将近用光之时，扩展为秒级时间或者毫秒级时间来挣得系统的移植时间窗口，其实只要扩展一位，完全可以再使用30年。

并发

对于中心服务器和REST发布方式，ID生成的过程涉及到网络IO和CPU操作，ID的生成基本都是内存到高速缓存的操作，没有IO操作，网络IO是系统的瓶颈。

相对于CPU计算速度来说网络IO是瓶颈，因此，ID产生的服务使用多线程的方式，对于ID生成过程中的竞争点time和sequence，我们使用concurrent包的ReentrantLock进行互斥。

机器ID的分配

我们将机器ID分为两个区段，一个区段服务于中心服务器发布模式和REST发布模式，另外一个区段服务于嵌入发布模式。

0-923：嵌入发布模式，预先配置，（或者由Zookeeper产生，第二版中实现），最多支持924台内嵌服务器。

924 – 1023：中心服务器发布模式和REST发布模式，最多支持300台，最大支持300*1万=300万/s的TPS.

如果嵌入式发布模式和中心服务器发布模式以及REST发布模式的使用量不符合这个比例，我们可以动态调整两个区间的值来适应。

另外，各个垂直业务之间具有天生的隔离性，每个业务都可以使用最多1024台服务器。

与Zookeeper集成

对于嵌入发布模式，服务启动需要连接Zookeeper集群，Zookeeper分配一个0-923区间的一个ID，如果0-923区间的ID被用光，Zookeeper会分配一个大于923的ID，这种情况，拒绝启动服务。

如果不想使用Zookeeper产生的唯一的机器ID，我们提供缺省的预配的机器ID解决方案，每个使用统一发号器的服务需要预先配置一个默认的机器ID。

注：此功能在第二版中实现。

时间同步

使用Linux的定时任务crontab，定时通过授时服务器虚拟集群（全球有3000多台服务器）来核准服务器的时间。

> **ntpdate -u pool.ntp.orgpool.ntp.org**

时间相关的影响以及思考：

1. 调整时间是否会影响ID产生功能？

> 未重启机器调慢时间，Vesta抛出异常，拒绝产生ID。重启机器调快时间，调整后正常产生ID，调整时段内没有ID产生。
>
> 重启机器调慢时间，Vesta将可能产生重复的时间，系统管理员需要保证不会发生这种情况。重启机器调快时间，调整后正常产生ID，调整时段内没有ID产生。

2. 每4年一次同步润秒会不会影响ID产生功能？

> 原子时钟和电子时钟每四年误差为1秒，也就是说电子时钟每4年会比原子时钟慢1秒，所以，每隔四年，网络时钟都会同步一次时间，但是本地机器Windows,Linux等不会自动同步时间，需要手工同步，或者使用ntpupdate向网络时钟同步。

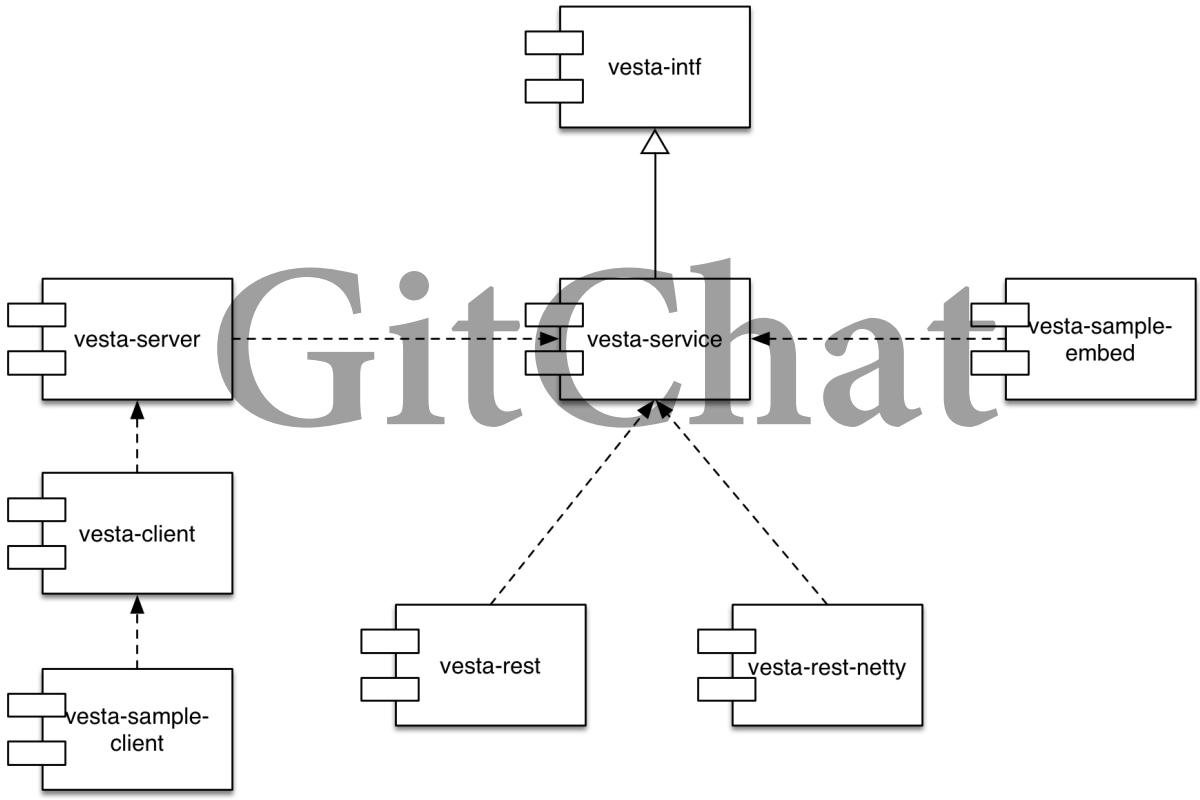
> 由于时钟是调快1秒，调整后不影响ID产生，调整的1s内没有ID产生。

多场景的发号器的实现方法

本节根据上一节的设计方案，介绍我们如何实现多场景发号器，并聚焦在项目结构、实现要点、并发处理等主题，并对关键的代码实现做注解。

项目组成结构

首先，我们的多场景发号器支持多种配置模式：嵌入发布模式、中心服务器发布模式、REST发布模式，因此，我们要对我们要实现的项目结构做个整体的规划，规划如下：



对应的项目结构如下：

```
/vesta-id-generator
/vesta-id-generator/vesta-client
/vesta-id-generator/vesta-doc
/vesta-id-generator/vesta-intf
/vesta-id-generator/vesta-rest
/vesta-id-generator/vesta-rest-netty
/vesta-id-generator/vesta-sample
/vesta-id-generator/vesta-server
/vesta-id-generator/vesta-service
/vesta-id-generator/vesta-theme
/vesta-id-generator/deploy-maven.sh
```

```
/vesta-id-generator/make-release.sh
/vesta-id-generator/pom.xml
/vesta-id-generator/LICENSE
/vesta-id-generator/README.md
```

对应的每个项目元素的职责和功能如下：

> 1. vesta-id-generator：所有项目的父项目。
> 2. vesta-id-generator/vesta-intf：发号器抽象出来的对外的接口。
> 3. vesta-id-generator/vesta-service：实现发号器接口的核心项目。
> 4. vesta-id-generator/vesta-server：把发号器服务通过dubbo服务导出的项目。
> 5. vesta-id-generator/vesta-rest：通过Spring Boot启动的REST模式的发号器服务器。
> 6. vesta-id-generator/vesta-rest-netty：通过Netty启动的REST模式的发号器服务器。
> 7. vesta-id-generator/vesta-client：导入发号器dubbo服务的客户端项目。
> 8. vesta-id-generator/vesta-sample：嵌入式部署模式和dubbo服务部署模式的使用示例。
> 9. vesta-id-generator/vesta-doc：包含架构设计文档、压测文档和使用向导等文档。
> 10. vesta-id-generator/deploy-maven.sh：一键发布发号器依赖Jar包到Maven库。
> 11. vesta-id-generator/make-release.sh：一键打包发号器。
> 12. vesta-id-generator/pom.xml：发号器的maven打包文件。
> 13. vesta-id-generator/LICENSE：开源协议，本项目采用Apache License 2.0。
> 14. vesta-id-generator/README.md：入门向导文件。

我们这样划分项目基于以下原则：

1. 我们开发的发号器适合多种用途、多种场景，我们不能简单的建设一个项目，把所有的需求都堆砌在一起，需要根据功能职责对项目进行划分，因此，我么主要将项目拆分成发号器服务的接口模块、发号器服务的实现模块、针对不同的发布模式的服务导出项目。

2. 我们开发的是一个开源的项目，希望开源项目简单实用，使用者下载后根据项目结构即可判断如何使用。因此，我们在跟项目中增加了README文档，以及更丰富的doc项目下的文档，并且提供了一键打包和发布的脚本，还提供了演示使用发号器项目的示例项目。

3. 我们分离了发号器的接口项目和实现项目，因为不同的场景的需求不一样，对于REST发布模式，不需要依赖发号器的接口和实现，对于dubbo服务的客户端只需要依赖发号器的接口即可，对于嵌入式发布模式，不但需要依赖发号器的接口还要依赖它的实现。

服务对外接口

根据我们前面小结对需求的整理，我们实现多场景发号器的接口如下：

```
public interface IdService {

    public long genId();

    public Id expId(long id);

    public long makeId(long time, long seq);

    public long makeId(long time, long seq, long machine);

    public long makeId(long genMethod, long time, long seq, long
machine);

    public long makeId(long type, long genMethod,  long time,
            long seq, long machine);

    public long makeId(long version, long type, long genMethod,
            long time, long seq, long machine);

    public Date transTime(long time);
}
```

其中主要包含如下的服务方法：

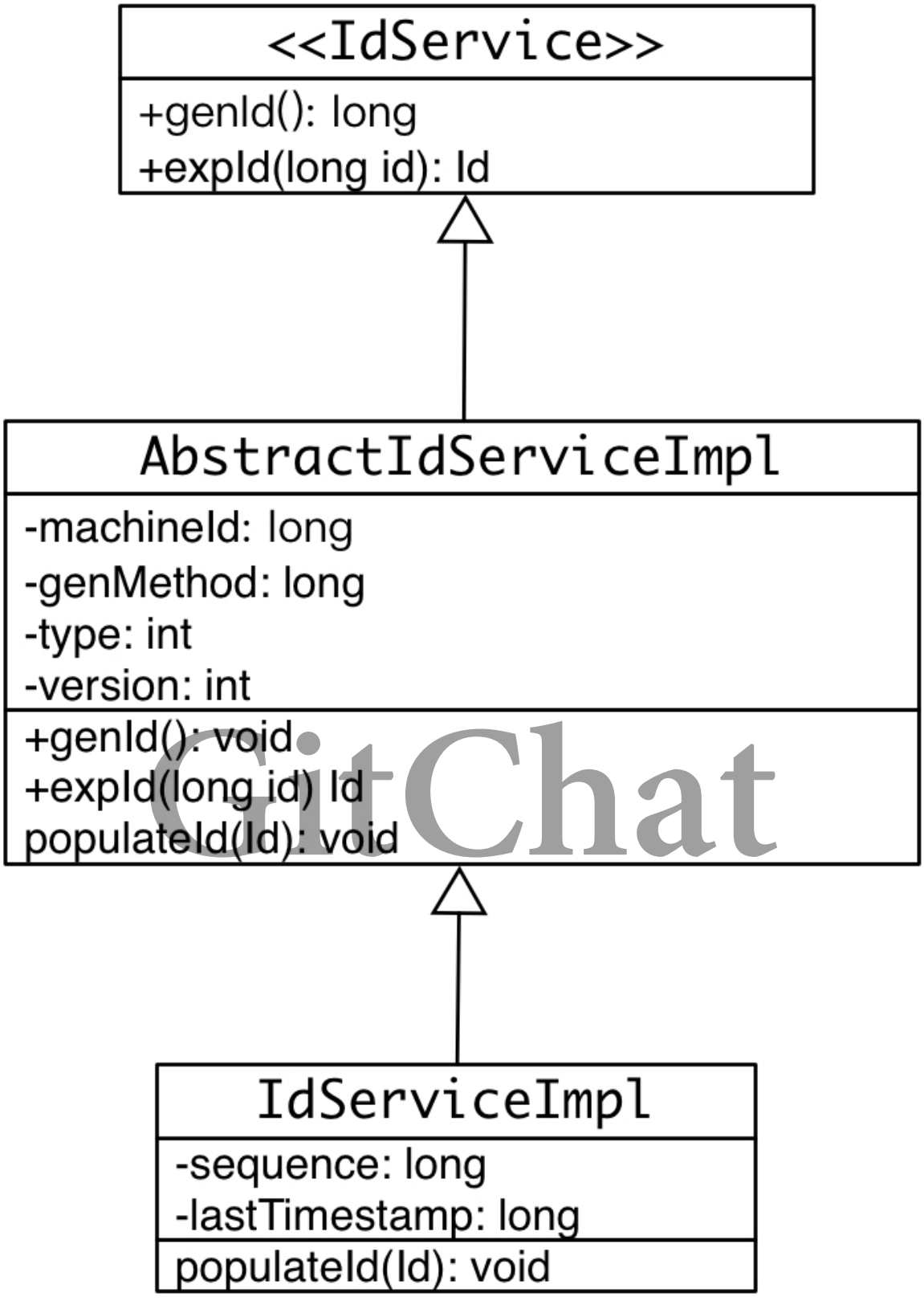
1. genId()：这是分布式发号器的主要API，用来产生唯一的ID。
2. expId(long id)：这是产生唯一ID的反向操作，可以对一个ID的内涵信心进行解读，用人类可读的形式来表达。
3. makeId(...)：用来伪造某一时间的ID。
4. transTime(long time): 这个方法是用来将整形时间翻译成格式化时间，是一个支持的方法。

从上面接口定义来看，简单、清晰、易懂，只定义必要的功能，方案按照重要程度排列。

核心实现方法

在实现类的设计上，我们设计了两层结构，抽象类AbstractIdServiceImpl和实体类IdServiceImpl，抽象类AbstractIdServiceImpl实现那些在任何场景下不变的逻辑，而可变的逻辑放到了实体类中实现，实体类IdServiceImpl则是最通用的实现方式。

实现类的类图如下：



从类图中看到，抽象类里包含了如下4个属性：

```
protected long machineId;
protected long genMethod;
protected long type;
protected long version;
```

这4个属性分别代表机器ID、生成方式、类型和版本，对于任何一个发号器部署实例，这些属性一旦固定下来将不会改变，因此，我们把这些属性和其处理逻辑放到了抽象的父类。

构成唯一ID的格式中的另外两个变量，时间和序列号，他们的产生方式是变化多端和多种多样的，因此，我们把着两个变量和处理他们的逻辑封装在子类，并且提供一个默认的实现子类。

时间和序列号属性如下：

```
private long sequence;
private long lastTimestamp;
```

现在我们来看产生发号器的逻辑，主逻辑封装在抽象父类中AbstractIdServiceImpl，代码如下：

```
public long genId() {
    Id id = new Id();

    populateId(id);

    id.setMachine(machineId);
    id.setGenMethod(genMethod);
    id.setType(type);
    id.setVersion(version);

    long ret = idConverter.convert(id);

    // Use trace because it cause low performance
    if (log.isTraceEnabled())
        log.trace(String.format("Id: %s => %d", id, ret));

    return ret;
}
```


我们清晰的看到，首先构造了一个Id元数据对象，然后调用了模板回调函数populateId，模板回调函数是一个抽象的方法：

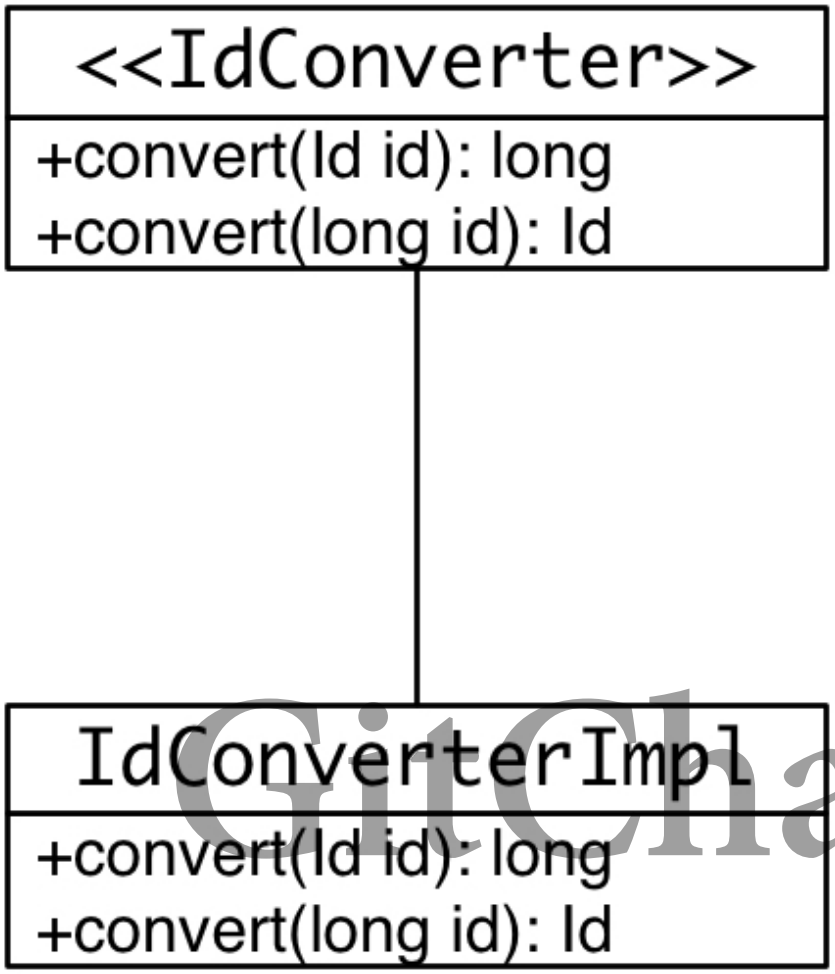
```
protected abstract void populateId(Id id);
```

这个抽象方法由子类来实现，子类根据不同的场景会有不同的实现，在这里我们只需要在父类中给子类进行处理的一个机会，子类主要负责根据某一算法生成唯一ID的时间和序列号属性，父类则将自己管理的属性机器ID、生成方式、类型和版本进行赋值，

ID格式的互相转换

主流程中在ID元数据对象中设置了ID的各个属性后，通过转换器类将ID的元数据对象转换成长整形的ID。

转换器类的设计如下：



转换器负责将ID元数据对象转换成长整形的ID，以及从长整形的ID转换成ID元数据对象，并且定义了清晰的转换接口，用来将来扩展，能够实现其他类型的转换。

ID元数据对象转换成长整形的ID的代码实现如下：

```
public long convert(Id id) {
    return doConvert(id, IdMetaFactory.getIdMeta(idType));
}

protected long doConvert(Id id, IdMeta idMeta) {
    long ret = 0;

    ret |= id.getMachine();

    ret |= id.getSeq() << idMeta.getSeqBitsStartPos();

    ret |= id.getTime() << idMeta.getTimeBitsStartPos();
```

```
        ret |= id.getGenMethod() << idMeta.getGenMethodBitsStartPos();

        ret |= id.getType() << idMeta.getTypeBitsStartPos();

        ret |= id.getVersion() << idMeta.getVersionBitsStartPos();

        return ret;
    }
```

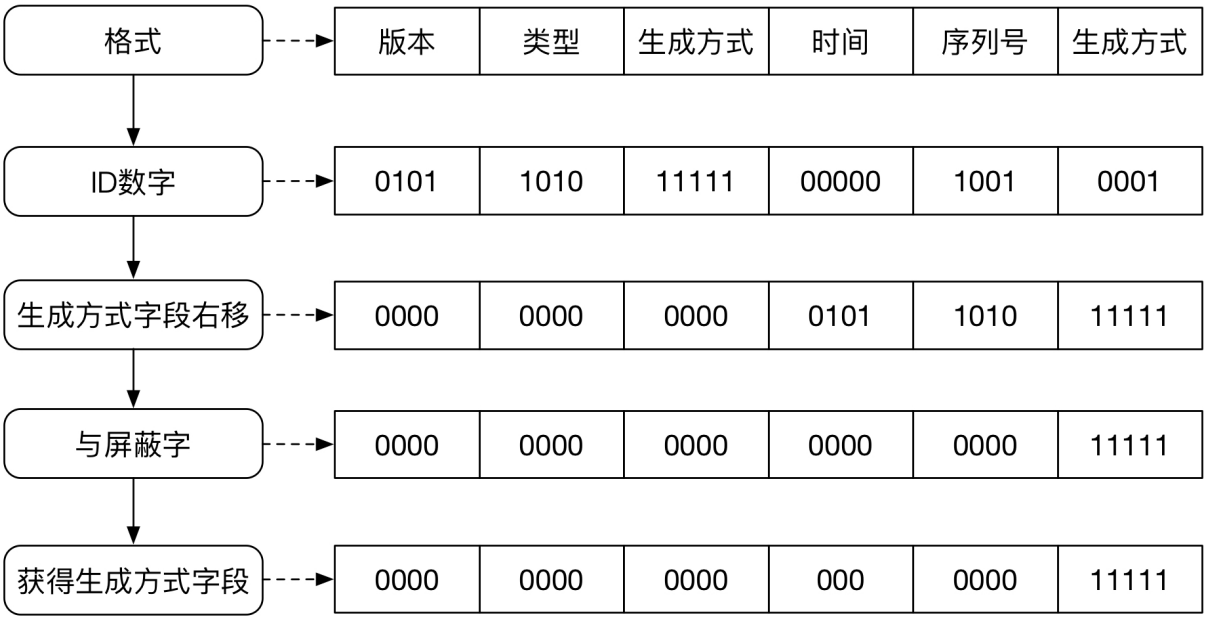
如上面代码实现，转换器根据ID元数据信息对象获取每个属性所在ID的位数，然后，通过左移来实现各个属性拼接到一个长整形数字里。

另外，我们前面接口设计中，有的时候我们需要从一个长整形的ID，解释成人类可读的格式，从中看到时间、序列号、版本、类型等属性，我们需要从长整形的ID转换成ID元数据对象，代码实现如下：

```
    public Id convert(long id) {
        return doConvert(id, IdMetaFactory.getIdMeta(idType));
    }

    protected Id doConvert(long id, IdMeta idMeta) {
        Id ret = new Id();

        ret.setMachine(id & idMeta.getMachineBitsMask());

        ret.setSeq((id >>> idMeta.getSeqBitsStartPos()) &
    idMeta.getSeqBitsMask());

        ret.setTime((id >>> idMeta.getTimeBitsStartPos()) &
    idMeta.getTimeBitsMask());

        ret.setGenMethod((id >>> idMeta.getGenMethodBitsStartPos()) &
    idMeta.getGenMethodBitsMask());

        ret.setType((id >>> idMeta.getTypeBitsStartPos()) &
    idMeta.getTypeBitsMask());

        ret.setVersion((id >>> idMeta.getVersionBitsStartPos()) &
    idMeta.getVersionBitsMask());

        return ret;
    }
```


请注意上面代码使用的是无符号右移操作，因为我们产生的ID包含的每一位二进制位都代表特殊的含义，没有数学上正负的意义，最左边的一位二进制也不是用来表示符号的。

另外，我们看到在做无符号右移操作的时候使用了屏蔽字，这是用来从ID数字中取出想要的某个属性的值，具体流程如下图：



我们举例说明，假设唯一ID数字包含的生成方式属性为1111，可以参考第二行，第三个方格，这个图只是一个示意图，每个属性的位数和设计不是不是一一对应的，现在我们想取出生成方式属性的数值1111。

首先，程序会把ID数字整体右移，直到生成方式属性位于最右端：

```
id >>> idMeta.getTypeBitsStartPos())
```


得到结果可参考如上图中第三行的数据。

然后，再与屏蔽字做与操作，得到结果为生成方式属性，参考上图的第五行的数据。

```
id >>> idMeta.getTypeBitsStartPos()) & idMeta.getTypeBitsMask()
```

屏蔽字参考上图中第4行，实现代码如下：

```
public long getGenMethodBitsMask() {
    return -1L ^ -1L << genMethodBits;
}
```

这里-1为64位全为1的二进制数字，首先将其左移属性所在位置的唯一，生成方式属性从右边开始的位置到数字最右面一位全为0，再与-1，也就是64位全为1的二进制数字做与操作，结果就形成了屏蔽字，参考上图中的第4行数据。

压测与性能保障

一款软件的发布必须保证满足性能需求，这通常需要在项目初期提出性能需求，在项目进行中做性能测试来验证，请参考本文末尾的源码连接下载源代码，查看性能测试用例，本章节只讨论性能需求和测试结果，以及改进点。

性能需求

最终的性能验证要保证每台服务器的TPS达到1万/s以上。

测试环境

由于笔者在自己使用的IBM Thinkpad X200的笔记本上做的测试，笔记本的CPU使用的Intel的双线程技术，机器设备比较老，性能比较低，因此，测试数据看起来并不是太高，但这些都是真实的数据，读者可根据实际情况使用更好的机器进行测试。

> 笔记本，客户端服务器跑在同一台机器
> 双核2.4G I3 CPU， 4G内存

嵌入发布模式压测结果

设置：

> **并发数：**100

测试结果：

测试	测试1	测试2	测试3	测试4	测试5	平均值/最大值
QPS	*431000*	*445000*	*442000*	*434000*	*434000*	*437200*
平均时间(us)	*161*	*160*	*168*	*143*	*157*	*157*
最大响应时间(ms)	*339*	*304*	*378*	*303*	*299*	*378*

中心服务器发布模式压测结果

设置：

> **并发数：**100

测试结果：

测试	测试1	测试2	测试3	测试4	测试5	平均值/最大值
QPS	*1737*	*1410*	*1474*	*1372*	*1474*	*1493*
平均时间(us)	*55*	*67*	*66*	*68*	*65*	*64*
最大响应时间(ms)	*785*	*952*	*532*	*1129*	*1036*	*1129*

REST发布模式（Netty实现）压测结果

设置：

> **并发数：**100
>
> **Boss线程数：**1
>
> **Workder线程数：**4

测试结果：

测试	测试1	测试2	测试3	测试4	测试5	平均值/最大值
QPS	*11001*	*10611*	*9788*	*11251*	*10301*	*10590*
平均时间(ms)	*11*	*11*	*11*	*10*	*10*	*11*
最大响应时间(ms)	*25*	*21*	*23*	*21*	*21*	*25*

REST发布模式（Spring Boot + Tomcat）压测结果

设置：

> **并发数：**100
>
> **Boss线程数：**1
>
> **Workder线程数：**2
>
> **Exececutor线程数：**最小25最大200

测试结果：

测试	测试1	测试2	测试3	测试4	测试5	平均值/最大值

QPS	*4994*	*5104*	*5223*	*5108*	*5100*	*5105*
平均时间(ms)	*20*	*19*	*19*	*19*	*19*	*19*
最大响应时间(ms)	*75*	*61*	*61*	*61*	*67*	*75*

性能测试总结

根据上面实测数据，我们得出如下的结论：

1. 根据测试，Netty服务可到达11000的QPS，而Tomcat只能答道5000左右的QPS。
2. 嵌入发布模式，也就是JVM内部调用最快，没秒可答道40万以上。可见线上服务的瓶颈在网络IO以及网络IO的处理上。
3. 使用Dubbo导入导出的中心服务器发布模式的QPS只有不到2000, 这比Tomcat提供的HTTP服务的QPS还要小，这个不符合常理，一方面需要查看是否Dubbo RPC需要优化，包括线程池策略，序列化协议，通信协议等，另外一方面REST使用apache ab测试，嵌入式发布模式使用自己写的客户端测试，是否测试工具存在一定的差异。
4. 测试过程中发现loopback虚拟网卡达到30+M的流量，没有到达千兆网卡的极限，双核心CPU占用率已经接近200%，也就是CPU已经到达瓶颈。

参考上面总结第三条，中心服务器的性能问题需要在后期版本跟进和优化，这块留给读者继续思考和实践。

完善的用户使用向导

Vesta多场景分布式发号器支持嵌入发布模式、中心服务器发布模式、REST发布模式，每种发布 模式的API文档以及使用向导可参项目主页的文档连接。

安装与启动

1. 下载最新版本的REST发布模式的发布包。

- 点击下载：

  > vesta-rest-netty-0.0.1-bin.tar.gz

- 如果你通过源代码方式安装Vesta的发布包到你的Maven私服，你可以直接从你的Maven私服下载此安装包：

  >

```
wget http://ip:port/nexus/content/groups/public/com/robert/vesta/vesta-rest-netty/0.0.1/vesta-rest-netty-0.0.1-bin.tar.gz
```

2. 解压发布包到任意目录。

- 解压：

```
tar xzvf vesta-rest-netty-0.0.1-bin.tar.gz
```

3. 解压后更改属性文件。

- 属性文件：

```
vesta-rest-netty-0.0.1/conf/vesta-rest-netty.properties
```

- 文件内容：

```
vesta.machine=1022
vesta.genMethod=2
vesta.type=0
```


注意：

1. 机器ID为1022, 如果你有多台机器，递减机器ID，同一服务中机器ID不能重复。
2. genMethod为2表示使用嵌入发布模式。
3. type为0, 表示最大峰值型，如果想要使用最小粒度型，则设置为1。

4. REST发布模式的默认端口为8088,你可以通过更改启动文件来更改端口号,这里以10010为例。

- 启动文件：

```
vesta-rest-netty/target/vesta-rest-netty-0.0.1/bin/server.sh
```

- 文件内容：

```
port=10010
```

5. 修改启动脚本，并且赋予执行权限。

- 进入目录：

```
cd vesta-rest-netty-0.0.1/bin
```

- 执行命令：

```
chmod 755 *
```

6. 启动服务。

- 进入目录：

```
cd vesta-rest-netty-0.0.1/bin
```

- 执行命令：

```
./start.sh
```

7. 如果看到如下消息，服务启动成功。

- 输出：

```
apppath: /home/robert/vesta/vesta-rest-netty-0.0.1
Vesta Rest Netty Server is started.
```


测试Rest服务

1. 通过URL访问产生一个ID.

- 命令：

```
curl http://localhost:10010/genid
```

- 结果：

```
1138729511026688
```

2. 把产生的ID进行反解。

- 命令：

```
curl http://localhost:10010/expid?id=1138729511026688
```

- 结果：

> **{“genMethod”:0,”machine”:1,”seq”:0,”time”:12235264,”type”:0,”version”:0}**

JSON字符串显示的是反解的ID的各个组成部分的数值。

3. 对产生的日期进行反解。

- 命令：

```
curl http://localhost:10010/transtime?time=12235264
```

- 结果：

> **Fri May 22 14:41:04 CST 2015**

4. 使用反解的数据伪造ID.

- 命令：

```
curl http://localhost:10010/transtime?time=12235264
```


- 结果：

> **1138729511026688**

总结思考

发号器作为分布式服务化系统不可或缺的基础设施之一，它在保证系统正确运行和高可用上发挥着不可替代的作用。而本文介绍了一款原创开源的多场景分布式发号器Vesta，并介绍了Vesta的设计、实现、以及使用方式，读者在现实项目中可以直接使用它的任何发布模式，既装既用，读者也可以借鉴其中的设计思路和思想，开发自己的分布式发号器，除了发号器本身，本文按照一款开源项目的生命周期构思文章结果，从设计、实现、验证到使用向导，以及论述遗留的问题等，并提供了参考的开源实现，帮助读者学习如何创建一款平台类软件的过程的思路，帮助读者在技术的道路上发展越来越好。