架构设计，偏理论，M2本阶段以架构设计为目标，预期时间为2年。

# Part 1 面向对象

Object-Oriented是一种程序设计方法，从现实世界中客观存在的事物出发来构造软件系统，尽可能运用人类的自然思维方式。

基本思想：使用对象、类、继承、封装、多态等基本概念来进行程序设计。

**OO VS Procedure-oriented**

都是模块化编程，Procedure-oriented把问题分割成**以功能函数为单位**的模块中，

Object-Oriented把问题抽象描述成**以对象为单位**的模块中。

### 理解面向对象

面向对象的不足

[Reference](https://mp.weixin.qq.com/s?__biz=MzA4NjExMjE2OQ==&mid=2650118787&idx=2&sn=decc1e219b637eee41e79188263babf3&chksm=87cc9845b0bb1153d5d59397bcf273677fe546c6daef82e8af66346ced0d4e6e81996d53609f&mpshare=1&scene=1&srcid=#rd])

### 三个基本特性

封装

客观事物封装成抽象的类，更符合人的自然思维。

封装数据结构和函数，隐藏细节(private/public),避免被外界破坏。

继承

多态

多态是IOC、模板模式实现的关键。

方法的动态绑定，实现运行时的类型决定对象的行为。多态的表现形式是父类指针或引用指向子类对象，在这个指针上调用的方法使用子类的实现版本。

C++多态是通过虚函数来实现的。

TODO4: invokevirtual执行过程

[Reference](https://blog.csdn.net/qq_35614059/article/details/79141866)

四大好处

维护性，扩展性，复用性，灵活性。

为什么说面向接口编程是面向对象设计的精髓？

这四大好处都依赖于面向接口编程。

### 1.7.3 五大基本原则（SOLID）

[Reference](https://blog.csdn.net/dongnan591172113/article/details/51996421)

需求总是变化，所以就需要根据准则设计软件，变化满足需求的同时还能保持软件内部的封装体系稳定，不被需求的变化影响。且使我们的代码稳健，易于复用，易于拓展，灵活优雅。

设计准则 VS 设计模式

不同的设计模式对应不同的需求，而设计原则代表永恒的灵魂，是更高层次的抽象。

#### 3.1.4.1 单一职责

Single Responseibility Principle(S)。一个类(模块)，一个功能, 不做多职能接口。职责过多，可能引起它变化的因素就越多，增加系统错误风险。

好处：解耦和增强内聚性。

#### 3.1.4.2 开放封闭原则

Open Closed Principle

核心思想：对修改封闭的，对扩展开放。

**具体实现**

通过继承和多态机制，重写/重载方法来改变固有行为，实现拓展，所以就是开放的。

对抽象编程，不对具体编程。因为抽象相对稳定，类依赖于固定的抽象，所以修改是封闭的。

#### 3.1.4.4替换原则

Liskov Substitution Principle

子类可以扩展父类的功能，但不能改变父类原有的功能。

继承作为面向对象三大特性之一，在给程序设计带来巨大便利的同时，也带来了弊端。比如使用继承会给程序带来侵入性，程序的可移植性降低，增加了对象间的耦合性（这里是父子类之间的耦合），如果一个类被其他的类所继承，则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，所有涉及到子类的功能都有可能会产生故障。

好处：保证继承复用是可靠地，违反必然导致违反开放封闭原则。

实现：面向接口编程

#### 3.1.4.3 接口隔离

Interface Segregation Principle

核心思想

使用多个小的专门的接口，而不使用一个大的总接口。接口应该是内聚的，避免“胖”接口。

一个类对另外一个类的依赖应该建立在最小的接口上，不要强迫依赖不用的方法，这是一种接口污染。  
    胖接口存在明显的弊端，会导致实现的类型必须完全实现接口的所有方法、属性等；而某些时候，实现类型并非需要所有的接口定义，而且对胖接口的修改将导致一连串的客户端程序需要修改

具体实现

分离的手段主要有以下两种：

1、委托分离，通过增加一个新的类型来委托客户的请求，隔离客户和接口的直接依赖，但是会增加系统的开销。

2、多重继承分离，通过接口多继承来实现客户的需求，这种方式是较好的。

分离后，每个类对应一个功能方法，将接口组装到类中，有个性化需求时，只要跟换新的实现类即可。

#### 3.1.4.5 依赖倒置

Dependency Inversion Principle

1.High-level modules should not depend on low-level modules. Both should depend on abstractions.（上层模块不应该依赖底层模块，它们都应该依赖于抽象。）

2.Abstractions should not depend on details. Details should depend on abstractions.

面向接口编程

依赖倒置是通俗讲就是面向接口编程，即对抽象编程，不对具体编程。因为抽象相对稳定的可有效控制模块（类）耦合关系，提高系统的可维护性以及可扩展性。

**倒置**



（图一） （图二）

图1依赖关系为高层依赖底层

图2都依赖于抽象后，低层次模块依赖于抽象接口，被倒置。

### 1.7.4 23种设计模式

不同的设计模式对应具体的需求，好处从SOLID准则的角度来答。总体分为三类：

* 创建型：工厂方法、抽象工厂、单例模式、建造者模式、原型模式。
* 结构型：适配器、装饰器、代理模式、外观模式、桥接、组合模式、享元模式。
* 行为型：策略模式、模板方法、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

#### 单例模式

[单例模式](http://blog.csdn.net/dongnan591172113/article/details/51027374)

非同步



延迟加载



Double Checked locking



* 1处volatile：new Singleton并非原子操作，instance为类变量，准备阶段



1.new给instance分配内存

2.invokespecial初始化对象

3.aload将instance指向分配的内存

JVM 的即时编译器中存在指令重排序的优化。最终的执行顺序可能是 1-2-3 也可能是 1-3-2

2处为什么判空?

防止阻塞Thread恢复后会多次new

#### 工厂模式

直接new对象是硬编码，几处可能引起变化

1.构造器参数修改，所有创建该类的代码都要修改。

2.修改接口实现需要到创建的地方修改，并不符合开闭原则，可能引起系统风险。

工厂模式将系统大量的对象创建工作封装到工厂类，将变化控制到最低，更符合开闭原则。

作用：

1）主要功能是实例化对象的

2) 集中管理对象生命周期。

好处: 创建对象的变化部分封装到工厂类，对象创建和使用解耦了。

InterfaceImp imp=Factory.getInstance();

Factory可根据XML配置定义子类。

简单工厂

对象创建封装到工厂类。工厂类本身不符合开闭原则。需求改变需要修改工厂类。不属于23种GOF设计

工厂方法

特点：每种对象有一个工厂。

1. IBusiness productA = **new** FactoryB().create();
2. productA.doService();
3. IBusiness productB = **new** FactoryB().create();
4. productB.doService();

当系统加入新产品，只要添加一个具体工厂和具体产品就可以了，系统的可扩展性也就变得非常好，符合开闭原则。

抽象工厂

同工厂方法模式的。只是它的一个工厂可以创建属于多种产品。

#### 策略模式

[Reference](http://www.runoob.com/design-pattern/strategy-pattern.html)

和工厂模式在结构上很相似，差别：

工厂关注对象创建，而策略模式**关注行为的封装**。

策略模式包含如下角色：

* Context: 环境类
* Strategy: 抽象策略类
* ConcreteStrategy: 具体策略类

1. **public** **class** Context {
2. **private** IStrategy strategy;
4. **public** Context(Strategy strategy){
5. **this**.strategy = strategy;
6. }
8. **public** **int** executeStrategy(**int** num1, **int** num2){
9. **return** strategy.doOperation(num1, num2);
10. }
11. }

对象创建的控制权在调用者，工厂模式中对象创建和工厂耦合

#### 观察者模式

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

观察者 VS 事件驱动

TODO3

#### 模版方法

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。

关注点分离

TODO https://www.cnblogs.com/asis/p/architecture-Soc.html

#### Facade

门面模式提供一个高层次的接口，使得子系统更易于使用。

“单一职责原则”，将一个系统划分为若干个子系统，降低了系统的复杂性。一个常见的设计目标是使子系统间的通信和相互依赖关系达到最小，而达到该目标的途径之一就是引入一个外观对象，它为子系统的访问提供了一个简单而单一的入口。

本质就是封装调用，MVC中的Control层调用Service甚至都可以理解为一种外观模式，将复杂的业务逻辑都卸载Service，Control作为Facade对象与外界（接口）对接。

再说一个使用环境，给外界提供接口的时候，可以只暴漏一个门面（接口对象），让外界都访问这个门面对象，门面对像内在做具体的业务逻辑进行区分是什么接口采用什么应对策略。

### 1.7.5 代码优化

在实际编程中，根据设计准则，总结如下几点：

* 对抽象编程，不对具体编程（面向接口）
* 开闭原则
* 封装变化
* 多用组合少用继承
* 使用继承时遵循里氏替换原则。

基本思想：尽可能的抽象，将变化控制到最小，做到高内聚、低耦合。

实现方法：使用对象、类、继承、封装、多态等基本概念来进行程序设计。

优化目标，什么样的代码是好代码？

答案：高内聚，低耦合。易替换，易扩展。

#### 低耦合&高内聚

模块独立性指每个模块只完成系统要求的独立子功能，并且与其他模块的联系最少且接口简单，两个定性的度量标准――耦合性和内聚性。

耦合性

指模块间相互联系紧密程度。为提高模块的独立性，模块间尽可能松散的耦合。

实现方法：依赖于抽象。

耦合性分类(低🡪高):

* 无直接耦合: 无调用关系，通过主模块的控制和调用产生联系。比如controller调用的service。

下面几种模块之间存在调用关系：

* 数据耦合: 传递简单的数据值，即值传递。
* 标记耦合: 传递的是数据结构，如数组名、类。
* 控制耦合: 传递的是控制变量（如开关、标志等）。
* 公共耦合: 访问同一个全局数据结构（包括单个变量），
* 内容耦合: 可能在汇编语言中出现。大多数高级语言都已设计成不允许出现。

1.高耦合度导致系统难维护，牵一发动全身，又会引入新的BUG。2.且不好拓展。

内聚性

模块内各元素（变量，语句、、、）联系的越紧密，则它的内聚性就越高。在模块划分时，遵循“一个模块，一个功能”的原则，尽可能使模块达到功能内聚。

内聚性分类(低🡪高)

* 偶然内聚: 模块内各元素之间没有任何联系。
* 逻辑内聚: 模块把几种相关的功能组合在一起，每次被调用时，由传送给模块参数来确定该模块应完成哪一种功能 。
* 时间内聚: 把需要同时执行的动作组合在一起。(无顺序)
* 过程内聚：必须以特定次序执行。（有顺序）
* 通信内聚:处理使用相同的输入数据或者产生相同的输出数据。
* 顺序内聚: 顺序+同一功能。前一功能元素输出就是下一功能元素的输入。（）
* 功能内聚: 顺序+同一功能 +不可再分

#### 解耦之面向接口编程

1. **public** **void** service() {
2. IBusiness business1 = Factory.create("A");
3. business1.doService();
4. }

依赖倒置准则：对抽象编程，不对具体编程。好处：有效控制模块（类）耦合关系。

依赖于抽象是面向对象设计的精髓，也是依赖倒置原则的核心。

#### 对象创建

* new：缺点：对象创建到处分散，如要更换，则需要大量修改代码。
* 工厂模式：调用的类和工厂有一定的耦合。
* 依赖注入：进一步解耦，

接口注入: bookA = (IBook)Class.forName("BookA").newInstance();

工厂模式将对象创建和使用解耦

好处

1.开闭原则：line

当然也可以传接口，这种把对象的创建推迟了，也可能推给调用者。

1. **public** **void** service(IBusiness business) {
2. business.doService();
3. }

#### 优化if-else

分支嵌套层次过深可读性下降（每次维护都要打开读）。维护时要改代码不符合开闭原则。当然，场景比较少用if-else更合适。

[Reference2](https://mp.weixin.qq.com/s?__biz=MzI3NzE0NjcwMg==&mid=2650123156&idx=1&sn=a917fde5e075223d610ac2a9e8e94656&chksm=f36bb6b5c41c3fa35dbe148b566c41f05cea7d5ce565b295aae96dfece70ecc09c4f360288ac&mpshare=1&scene=1&srcid=#rd])

**1)接口分层**

1. **public** **void** handleImpl(Item item) {
2. **if** ("saved".equal(item.getStatus())) {
3. processSavedStatus();
4. }**else** **if**("finished".equal(item.getStatus())){
5. processFinishedStatus();
6. }
7. }

不同逻辑剥离到对应方法，看着清爽多了，判空同样可以放在外部接口只处理一次

1. **public** **void** handle(Item item) {
2. **if** (item == **null**) {
3. **return**;
4. }
5. handleImpl(item);
6. }

**2）工厂模式**

从工厂里取实现类。

1. **public** **void** handleImpl(String type) {
2. TargetExecutor executor=Fatory.getExecutor(type);
3. executor.process();
4. }

工厂模式难免又引入分支，使用Map替代分支语句，对工厂模式进一步优化

1. **private** **void** init() {
2. map.put(TYPE\_LINK, Link.**class**);
3. map.put(TYPE\_IMAGE, Image.**class**);
4. }
6. **public** ShareItem createShareItem(**int** type) {
7. Class<? **extends** ShareItem> shareItemClass = map.get(type);
8. **return** shareItemClass.newInstance();
9. }

#### java bean

成员变量类型用基本类型还是boxed类型？

看场景，尽量使用 unboxed 类型,需要使用对象的时候用 boxed 类型，尽可少的进行 boxed <==> unboxed 转换。

# Part 2 架构分析

软件架构优劣的衡量要素：

1.性能 2.可用性 3.伸缩性 4.扩展性 5.安全性

(对前几章总结)

### 3.2.1.1 性能

#### 性能指标

* Concurrency：并发数。能同时处理的用户数。高并发
* Response：响应时间。发出请求到收到相应数据的时间
* 吞吐量：系统单位处理的请求数。QPS（query per second）
* 性能计数器：包括System Load、对象和线程数、内存、CPU、磁盘、网络IO

PV：page view/浏览量/点击量。默认指24小时内。同一user浏览同一页面不重复计算PV量。

UV：unique visitor，独立IP访问量。

TODO Nginx 统计PV [Reference](https://www.cnblogs.com/lianzhilei/p/6018421.html)

如何统计实际的QPS？

#### 性能测试方法

* 性能测试：性能预期
* 负荷测试：找到性能安全临界线
* 压力测试：超过安全符合继续施压，获得最大压力承受能力

### 3.3.4 可用性

Reads and writes always succeed，即服务一直可用，而且是正常响应时间。高可用的架构在系统中任何一台或多台服务器宕机及其他问题，系统依然可用。

不可用原因：普通商用服务器+服务器数量大=宕机概率大

度量指标

|  |  |  |
| --- | --- | --- |
| Avalilability | 宕机时间（year） | Score |
| 99% | 88h | 基本可用 |
| 99.9%(3个9) | 8.76h | 较高可用 |
| 99.99%(4个9) | 52min | 有自动恢复能力的高可用 |
| 5个9 |  | 极高可用 |

计算公式 [Reference](https://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247487977&idx=1&sn=f893a169f8232bf08035ac1ed260a089&chksm=e9c5e858deb2614e9c94f2c2d0e0da9b99fd76f98ad5395f12430c427bb36b7633516a2fbf81&mpshare=1&scene=1&srcid=#rd])

#### 提升可用性

1）冗余

* Application Server: 多台服务器通过负载均衡组成集群，服务失效转移
* 存储服务: 数据实时冗余备份，宕机时数据访问转移到可用Server，并进行数据恢复。

2）减少故障上线

预发布验证 自动化测试 自动化发布 灰度发布

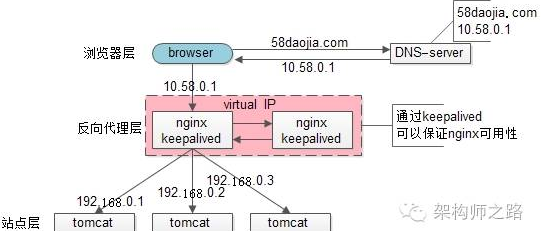
#### 可用性服务

Nginx：负载均衡组成集群，且进行无状态服务即不保存Session。

TODO [亿级PV的负载均衡](https://mp.weixin.qq.com/s?__biz=MzU0OTE4MzYzMw==&mid=2247485829&idx=1&sn=fa04dbd3fec75195232cafbc0085084d&chksm=fbb2807bccc5096d1f66c129f5a9ae1288e2b76aa0b1bf16e807c924af6529abd9d4f46e3261&mpshare=1&scene=1&srcid=#rd])

1.Nginx+keepalived高可用

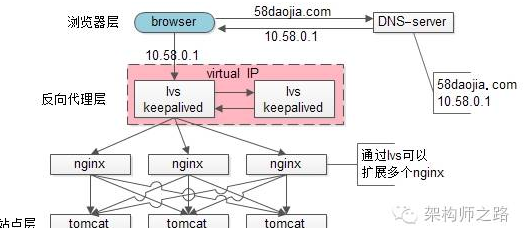
keepalived：检测服务状态存活性，常用来做高可用



当一台nginx挂了，keepalived能够探测到，并将流量自动迁移到另一台nginx上。

单点接入吞吐量有上线。

2.lvs+keepalived



* lvs实施在操作系统层面；f5性能更好在硬件层面；它们性能比nginx好很多。大部分公司到这一步基本就能解决接入层高可用、扩展性、负载均衡的问题。
* lvs也有性能上限，但没几个公司要考虑这个问题。

#### 可用性数据

1）冗余备份

2）失效转移

失效确认，访问转移，数据恢复

数据恢复：

### 3.3.2 伸缩性

随着不断上升的并发访问压力和数据增长，需要不断向集群加入服务器的手段来缓解。伸缩性衡量是否方便构建集群，是否容易添加服务器。

#### 应用服务器

负载均衡不仅提升网站的可用性，还提升伸缩性。

#### 数据存储服务器

不同于缓存，数据存储服务器集群的伸缩性对data的持久化和可用性提出更高要求。

关系数据库

* 主从复制：数据复制，进行简单伸缩。Master写数据，Slave读数据实现读写分离。
* 分库：受跨库join制约。
* 分表：基本主从复制分库，一些单表数据量过大。将一张表拆分开分表存到多个db。

#### NoSql

缓存设计

缓存数据丢失可从db获取，不影响业务，而db需要保证服务可用和数据可靠存储。

MongoDB

易伸缩，自动故障转移。易伸缩指的是提供了分片能力，能对数据集进行分片，数据的存储压力分摊给多台服务器。自动故障转移是副本集的概念，MongoDB能检测主节点是否存活，当失活时能自动提升从节点为主节点，达到故障转移。

### 3.3.3 扩展性

指现有系统影响最小的情况下，系统扩展或提升的能力。开闭原则。

分布式系统集成方式：1）分布式消息队列 2）分布式服务

**消息中间件**

通过消息分解系统解耦性，不同子系统处理同一消息

**分布式服务**

具体内容，见微服务章节。

### 3.2.5 安全性

TODO

js Cookie窃取和Session劫持

## 4.2 性能优化

从Http建立请求到response，整个生命周期都有优化的空间。宗旨是减少资源消耗和不合理的使用。

### 2.5.1 WEB系统优化

前端优化

* 合并css、js，减少Http请求
* 启用压缩，减少数据传输
* 使用浏览器缓存。生成新的js时，只需更新Html中引用。
* CSN加速：缓存静态资源，如图片、文件、CSS、JavaScript脚本、静态网页等。
* 反向代理

应用服务器

系统架构

* 缓存：加快数据读取，降低DB访问压力。网站遇到性能瓶颈，第一个想到的解决方案
* 异步操作：加快请求响应及实现削峰。
* 使用集群：使用负载均衡，有更好的响应延迟，提高吞吐能力。

代码优化

* Thread：多线程,最大发挥CPU和处理IO阻塞。
* IO：使用NIO(非阻塞，基于事件驱动)
* GC：减少垃圾回收。
* 锁优化：尽量不使用锁,减小锁的力度。
* 资源复用，比如单例和线程池。
* 合理的数据结构

Protocol

HTTP 1.1

**最大并发数限制**

并发指建立的TCP连接数。可以一个连接发送多个请求。Server端按发送顺序处理。

### 3.5.3 性能监测

* 性能测试：性能预期
* 负荷测试：找到性能安全临界线
* 压力测试：超过安全符合继续施压，获得最大压力承受能力

[压力测试](https://blog.csdn.net/wyingquan/article/details/414608)

#### 2.2.1.3 测试流程

很多时候在线上出现问题时，往往会伴随着某些指标的异常。大部分情况下，在问题发生之前，某些指标就会提前有异常显示。

step 1 概括

top查看整个系统负荷，正常情况单核负荷在0.7，四核就是3一下。

##### 7.2.2 系统卡顿

系统卡顿，响应慢，往往会伴随一些系统指标异常，第一步应找到系统瓶颈。可能在CPU，IO，也可能在内存。一些常见引起卡顿的原因

* 操作数据库 %idle值高
* 大量计算 %idle值低，
* 大量IO。%idle值高，%iowait值高，详细信息看iostat
* 死锁：大量线程执行同一方法？方法可能有request请求
* 大量频繁线程切换
* 内存：内存分配

定位线程

ps或jps找java进程pid🡪top -Hp $pid H 查看耗cpu时间最多的几个线程🡪线程ID转换成16进制🡪jstack thread dump

场景分析

CPU使用率高，响应慢

CPU持续飙高，定位到占用CPU最大的线程分析。可能的原因，

CPU处理能力相对较低/计算量大。

CPU使用率不高，响应慢

特征：%idle值高，dump线程，分析线程stuck原因。

**IO**：%iowait值高。分析IO，详细信息看iostat

**数据库**

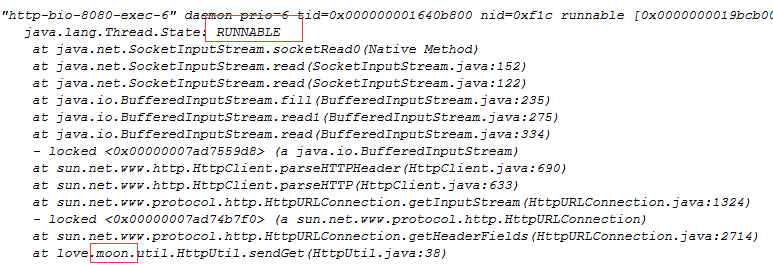
**内存**：内存分配

线程切换：

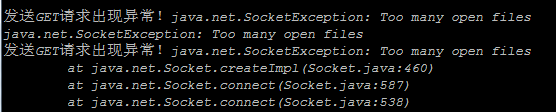
请求无响应

各项指标正常。是否所有的Runnable线程都一直在执行相同的方法，如果是的，恭喜你，锁住了！

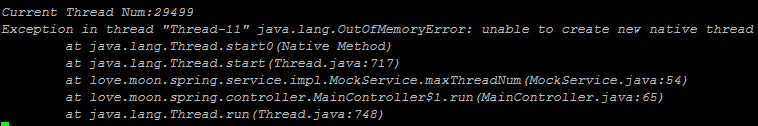
TODO线程池 大量runnable



请求第三方，如果不返回，线程一直RUNNABLE



超过4000个连接后



#### 附件 性能计数器

包括System Load、对象和线程数、内存、CPU、磁盘、网络IO。下面从不同角度解释。

##### Linux

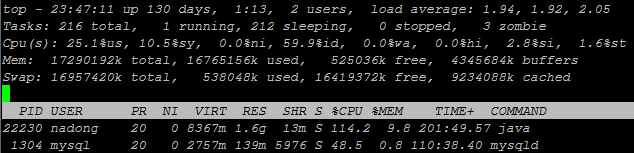
todo

strace

sar

###### top

动态实时显示。关注%CPU %MEM



**Cpu(s)**

%us user CPU time，这里指整个CPU计算能力的百分比。标题中%CPU指单核的。

%ni：改变过优先级的进程； %id：idle； %wa ：I/O waiting

%hi：硬中断，%si：软中断，%st: steal time

**Mem**

used：程序使用的+buffers. centos 7不包括缓存。

free：未分配

**S**：R=运行 S=睡眠 D=不可中断的睡眠状态T=跟踪/停止 Z=僵尸进程

快捷键**：**P:CPU排序。M:内存排序，H: show thread，1:show cups 列出每个cpu负荷

Title:

VIRT 进程使用的虚拟内存总量，单位kb。VIRT=SWAP+RES

**Load Average**

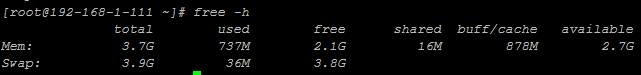
一段时间 (1 分钟、5分钟、15分钟) 内平均 Load

系统负荷正常范围：

* 大于0.7，必须开始调查了，问题出在哪里，防止情况恶化。
* 持续大于1.0，必须动手寻找解决办法，把这个值降下来。
* 当系统负荷达到5.0，就表明系统有很严重的问题，长时间没有响应，或者接近死机了。

以上指标基于单CPU的，四核机器的负载最好保持在3(4\*0.7 = 2.8)以下。

###### free



used 程序使用的

free：未分配

avaliable：free+buff/cache-不可回收部分（tmpfs）

###### netstat

1.Linux下查看并发连接数，一个浏览器可能建立几个连接，非并发用户数

netstat -na | grep ESTABLISHED | grep 8080 | wc -l

###### pidstat

检测CPU，内存，IO，线程切换等。每次运行统计的时间起点为上次运行时间。

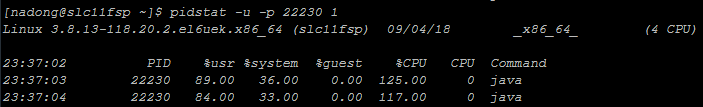
–p pid.

-u/r/d cpu/内存/ io. 默认显示CPU信息.

pidstat 2 10显示间隔为2s，共显示10次。

-u

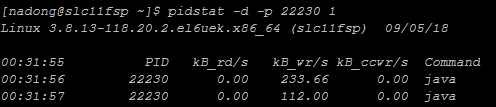
统计CPU



CPU:在那个核运行

针对单个的线程stat，和top类似

-d



###### iostat

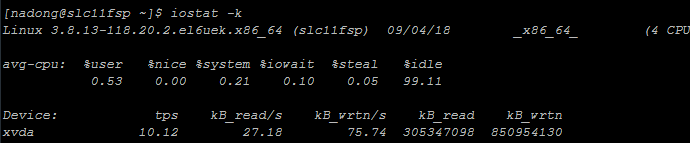
统计CPU和IO。针对整个系统的，不能对单个线程统计，区别于top/pidstat。

[Reference](https://linuxtools-rst.readthedocs.io/zh_CN/latest/tool/iostat.html)

-k/m 以kb/M为单位显示

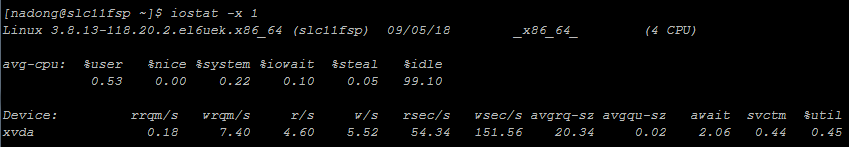
-C/d 显示CPU/磁盘 使用情况

-d IO



TPS 每秒I/O请求次数，可理解为吞吐量。（一次传输的大小未知）

-x 详细信息。



r/s: 每秒完成的读 I/O 设备次数。即 rio/s

w/s: 每秒完成的写 I/O 设备次数。即 wio/s

avgqu-sz: 平均I/O队列长度。

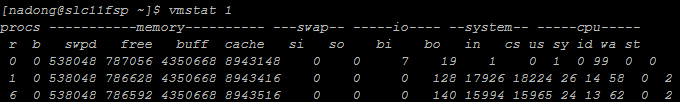
await: 响应时间，平均每次设备I/O操作的等待时间 (毫秒)。

svctm: 平均每次设备I/O操作的服务时间 (毫秒)。

%util拼接每次I/O消耗的cpu百分比

###### vmstat

[Reference](https://linuxtools-rst.readthedocs.io/zh_CN/latest/tool/vmstat.html)



**proces**: r 分配到CPU进程数，b阻塞进程

**memory**: swpd虚拟内存已用大小。

**Swap**: si 每秒从交换区写到内存的大小，so，每秒写入交换区的内存大小

io

单位：blocks/s，Linux块的大小为1M。从内存角度，bi 从block device 输入, bo 输出到device,

system

in中断数/每秒，cs 上下文切换/s。context指process context

**cpu**: us,sy,id,wa

##### JDK

###### jstack

生成线程快照。

[Reference](http://www.hollischuang.com/archives/110)

**用法**：jstack <pid>

**主要目的**：定位线程长时间停顿原因，如线程间死锁、请求外部资源导致的长时间等。

Thread Dump分析

prio：默认值为5，最大值为10，最小值为1 os\_prio ，tid：线程id，nid：native id，操作系统记录的id，thread state

RUNNABLE

locked <地址> (目标)。进入synchronized方法, 申请对象锁成功才会出现此句



BLOCKED(on object monitor)

waiting to lock <地址> (目标)：使用synchronized申请对象锁未成功,在迚入区等待。

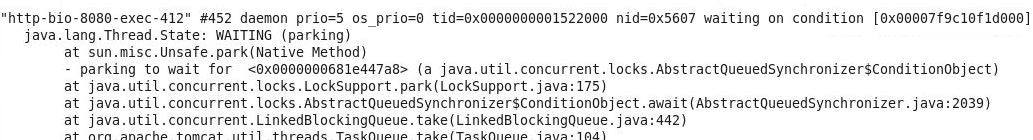


WATTING(on object monitor)

waiting on <地址> (目标)。synchronized修饰的方法调用wait()。Thread释放锁后会**等待锁**



WATTING(parking)

- parking to wait for <地址> (Object)

TIMED\_WAITING(parking)



这里tomcat线程池minSpareThreads设置较大，大量线程将parking

TIMED\_WAITING(sleeping)/ TIMED\_WAITING(on object monitor)

at java.lang.Thread.sleep

**Thread State**

Wait on condition

死锁

http://www.hollischuang.com/archives/2409

http://www.hollischuang.com/archives/110

###### jstat

jstat -option pic < interval >

**gcutil**



S0：年轻代中第一个survivor（幸存区）已使用的占当前容量百分比   
S1：年轻代中第二个survivor（幸存区）已使用的占当前容量百分比   
E：年轻代中Eden已使用的占当前容量百分比   
O：old代已使用的占当前容量百分比

P：perm代已使用的占当前容量百分比   
YGC：从应用程序启动到采样时年轻代中gc次数   
YGCT：从应用程序启动到采样时年轻代中gc所用时间(s)

FGC：从应用程序启动到采样时old代(全gc)gc次数   
FGCT：从应用程序启动到采样时old代(全gc)gc所用时间(s)   
GCT：从应用程序启动到采样时gc用的总时间(s)

**gc**



 S0C：年轻代中第一个survivor（幸存区）的容量 (byte)   
 S1C：年轻代中第二个survivor（幸存区）的容量 (byte)   
 S0U：年轻代中第一个survivor（幸存区）目前已使用空间 (byte)   
 S1U：年轻代中第二个survivor（幸存区）目前已使用空间 (byte)   
 EC：年轻代中Eden（伊甸园）的容量 (byte)   
 EU：年轻代中Eden（伊甸园）目前已使用空间 (byte)   
OC：Old代的容量 (byte)   
OU：Old代目前已使用空间 (byte)   
PC：Perm(持久代)的容量 (byte)   
PU：Perm(持久代)目前已使用空间 (字节)

**gcnew**

年轻代对象的信息



**gcold**

 old代对象的信息



**gcoldcapacity**

old代对象的信息及其占用量



OGCMN：old代中初始化(最小)的大小 (字节)   
OGCMX：old代的最大容量 (字节)   
OGC：old代当前新生成的容量 (字节)   
OC：Old代的容量 (byte)

**gcnewcapacity:**



 NGCMN：young初始化(最小)的大小 (字节)   
 NGCMX：年轻代(young)的最大容量 (字节)   
 NGC：年轻代(young)中当前的容量 (字节)

S0CMX：年轻代中第一个survivor（幸存区）的最大容量 (字节)   
S1CMX ：年轻代中第二个survivor（幸存区）的最大容量 (字节)

ECMX：年轻代中Eden（伊甸园）的最大容量 (字节)   
**gcpermcapacity**

 PGCMN：perm代中初始化(最小)的大小 (字节)   
  PGCMX：perm代的最大容量 (字节)     
  PGC：perm代当前新生成的容量 (字节)   
PC：Perm(持久代)的容量 (byte)

###### jmap

dump

jmap -dump:format=b,file=/scratch/tools/heamdump.out 22204

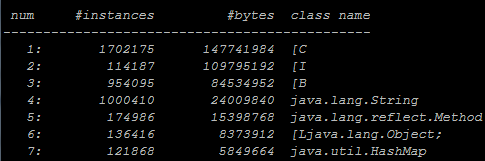
live：dump only live objects; if not specified,all objects in the heap are dumped.

format=b：binary format

-histo

jmap –histo[:live] pid

堆对象信息



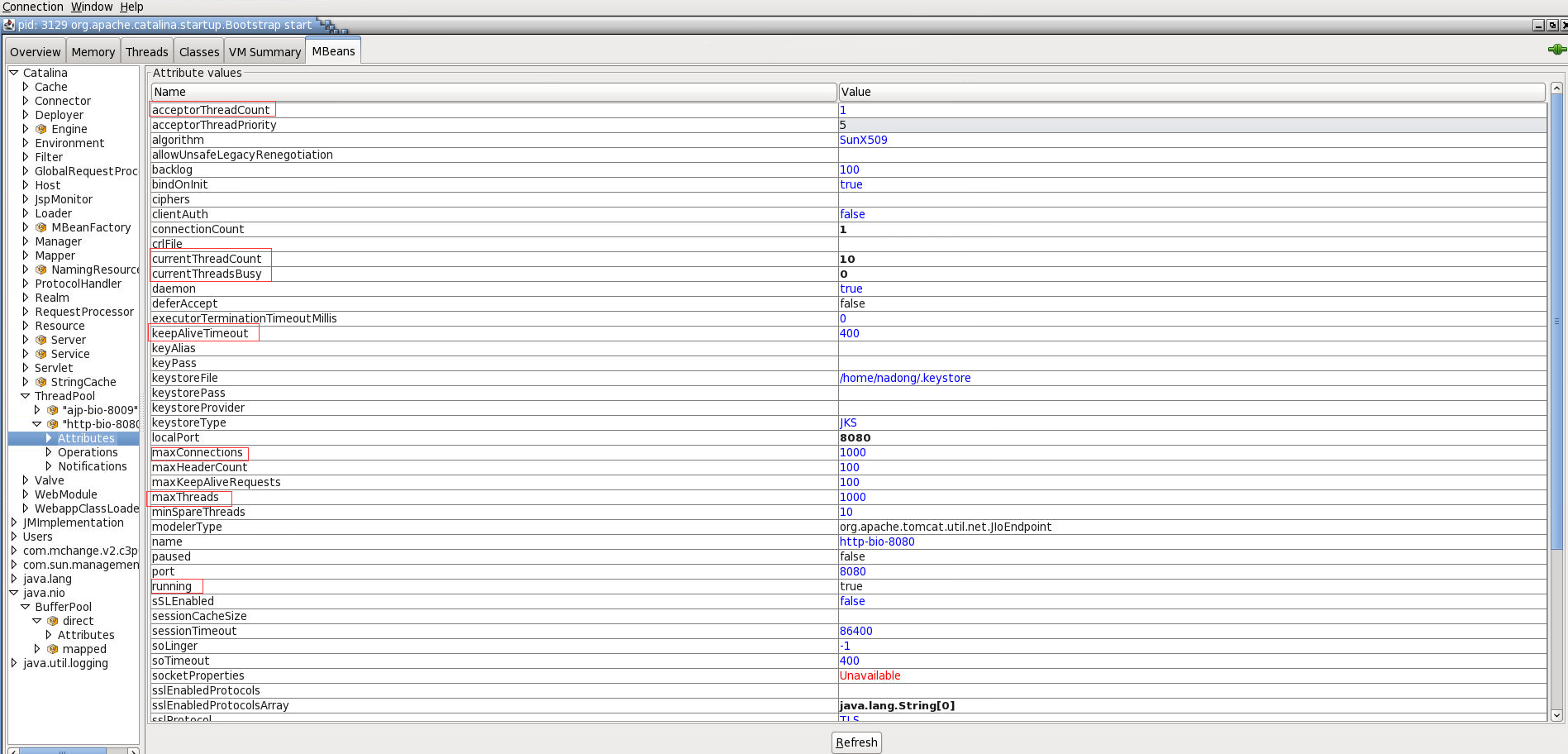
jconsole

配置

JAVA\_OPTS

-Djava.rmi.server.hostname=127.0.0.1 -Dcom.sun.management.jmxremote.port=8090 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false

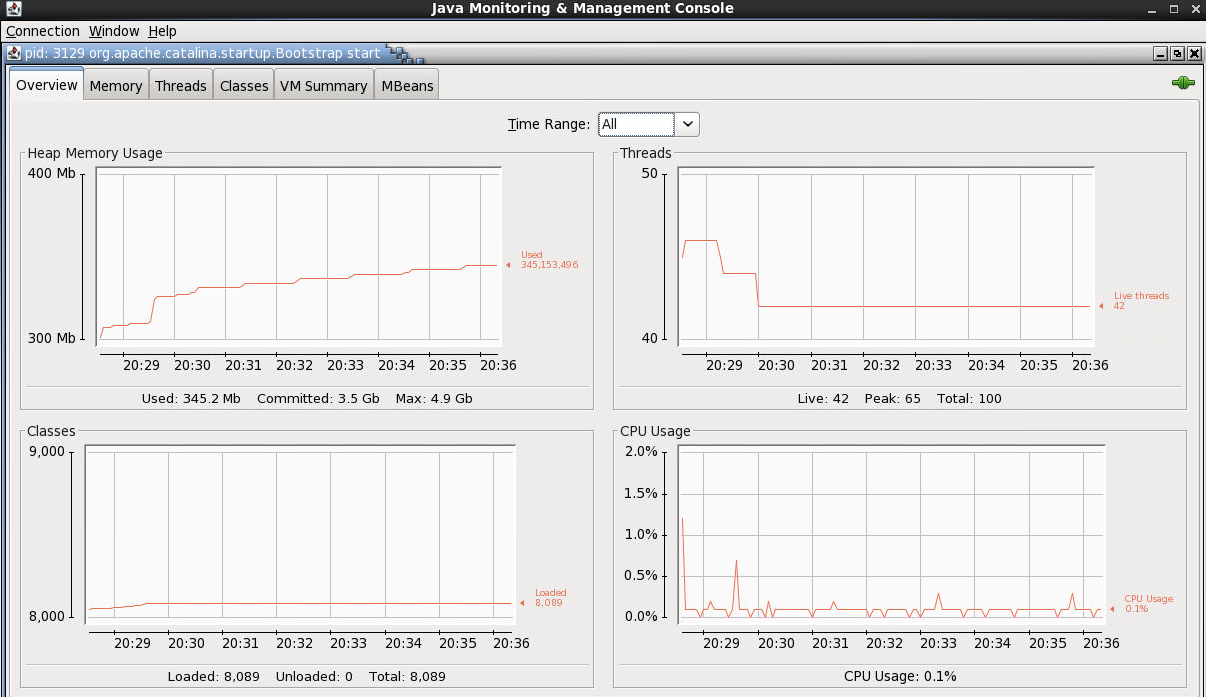
查询Acceptor属性



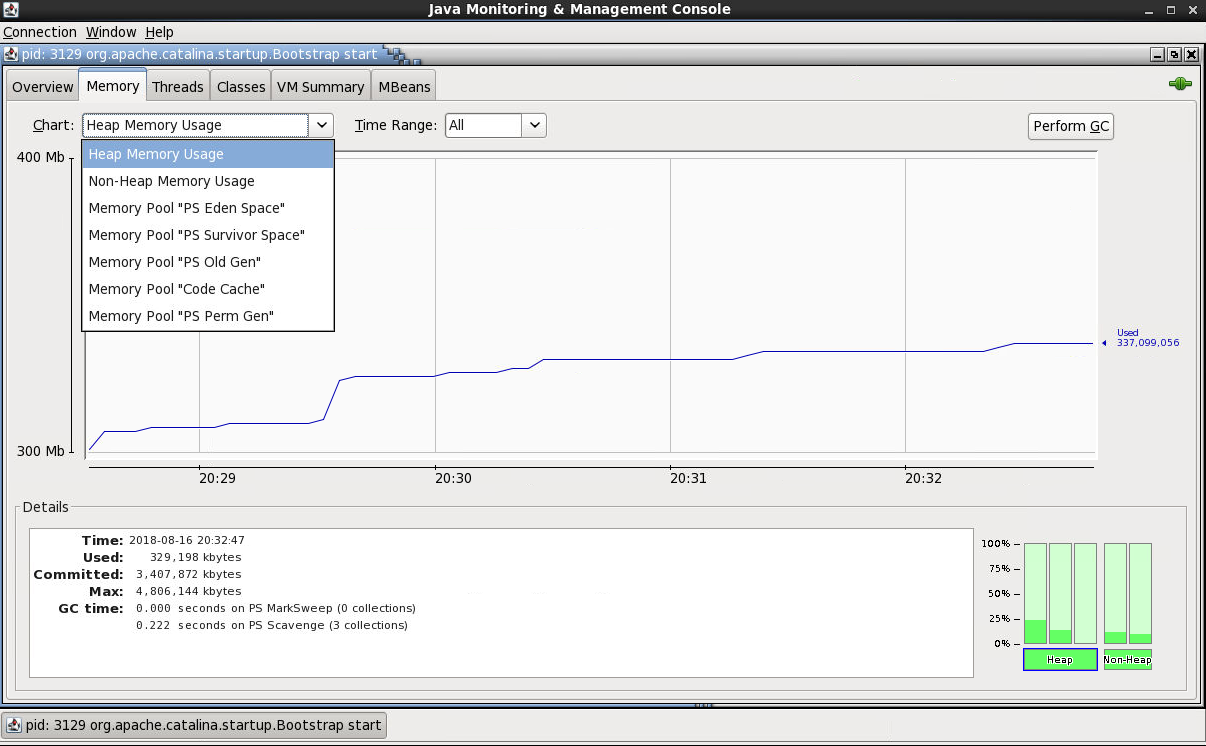
currentThreadCount

currentThreadBusy 线程池

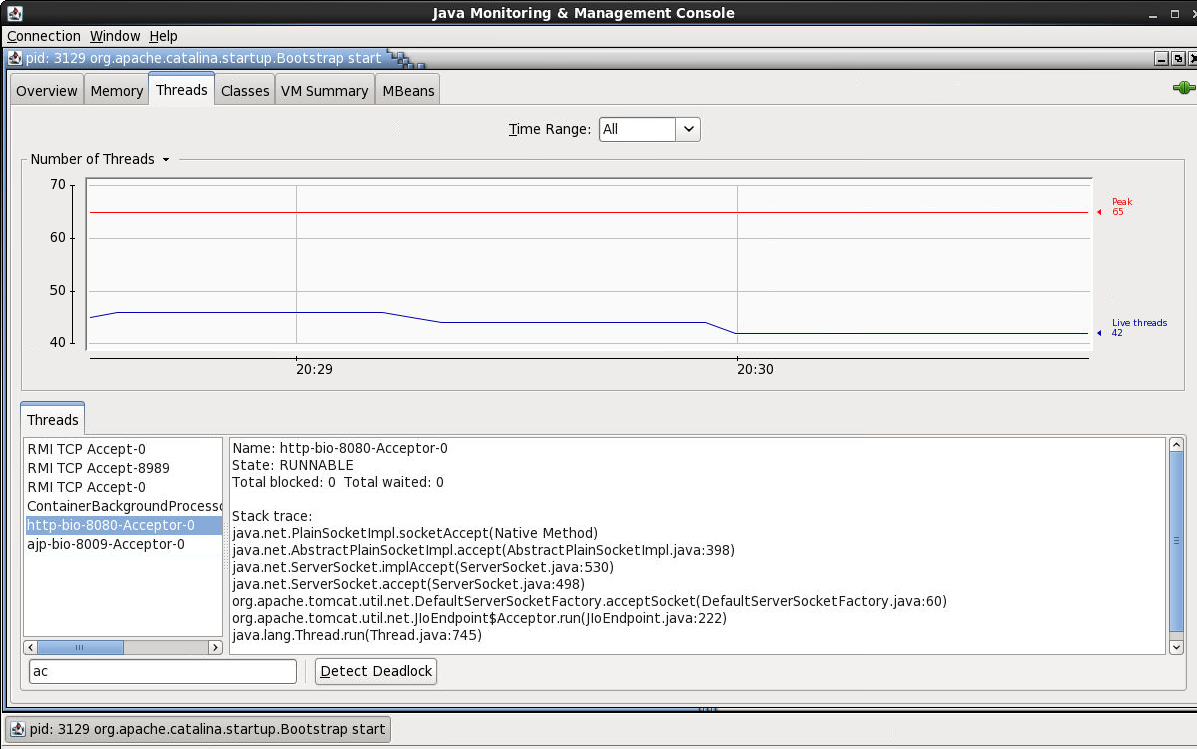
Overview



Memory



Thread



Total blocked

Total waited:

**state**

Wait on condition

Jvisual

JDK自带性能检测工具



**Running**: thread is still running.   
**Sleeping**: thread is sleeping (method yield () was called on the thread object)  
**Wait**: thread was blocked by a mutex or a barrier, and is waiting for another thread to release the lock

**Park**: parked threads are suspended until they are given a permit. Unparking a thread is usually done by calling method unpark () on the thread object  
**Monitor**: threads are waiting on a condition to become true to resume execution

##### Tomcat Mgr

[Reference](https://tomcat.apache.org/tomcat-7.0-doc/manager-howto.html)

Server Status

JVM

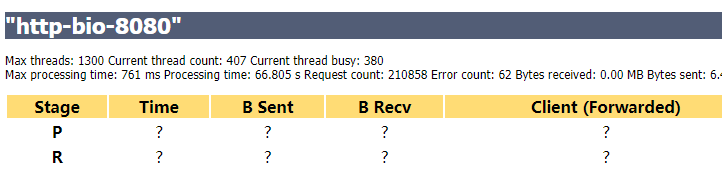
Free:可用内存

Total：当前分配。包括Free和未使用

Max：最大:

Thread Pool

<http://localhost:8080/manager/status>



Current thread busy: Runnable状态的线程

**Stage**

P: Parse and prepare request 正在处理请求

R: Ready. 线程sleep状态

S: Service

F: Finishing

K: Keepalive

TODO2 linux 命令 tomcat 每次只能accept一个request？

Status

<http://localhost:8080/manager/status?XML=true>

Thread Dump

<http://localhost:8080/manager/text/threaddump>

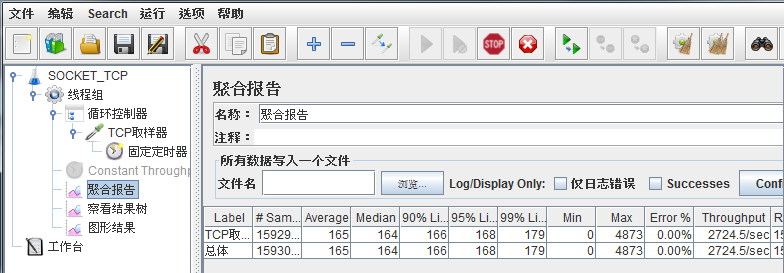
VM Info

<http://localhost:8080/manager/text/vminfo>

<http://localhost:8080/manager/status?XML=true>

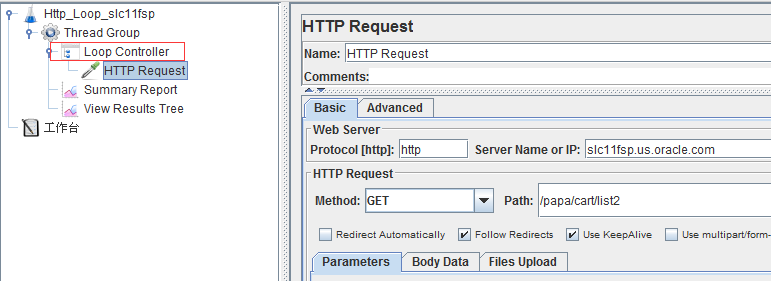
##### JMeter

TCP



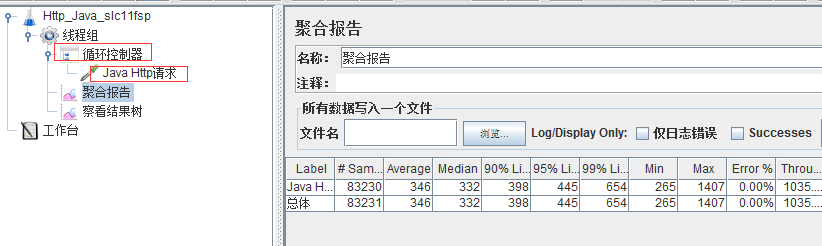
HTTP

短连接



此处是自带的HTTP Request，虽然Loop，但每个request都会重新建立TCP连接，使用完close。

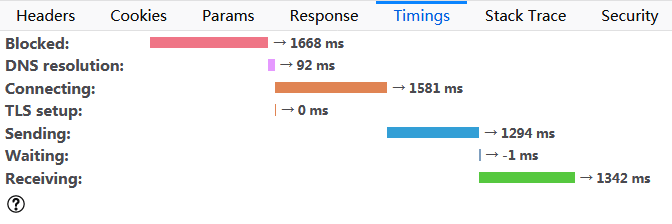
长连接



java中不要close连接

##### Browser

FireFox



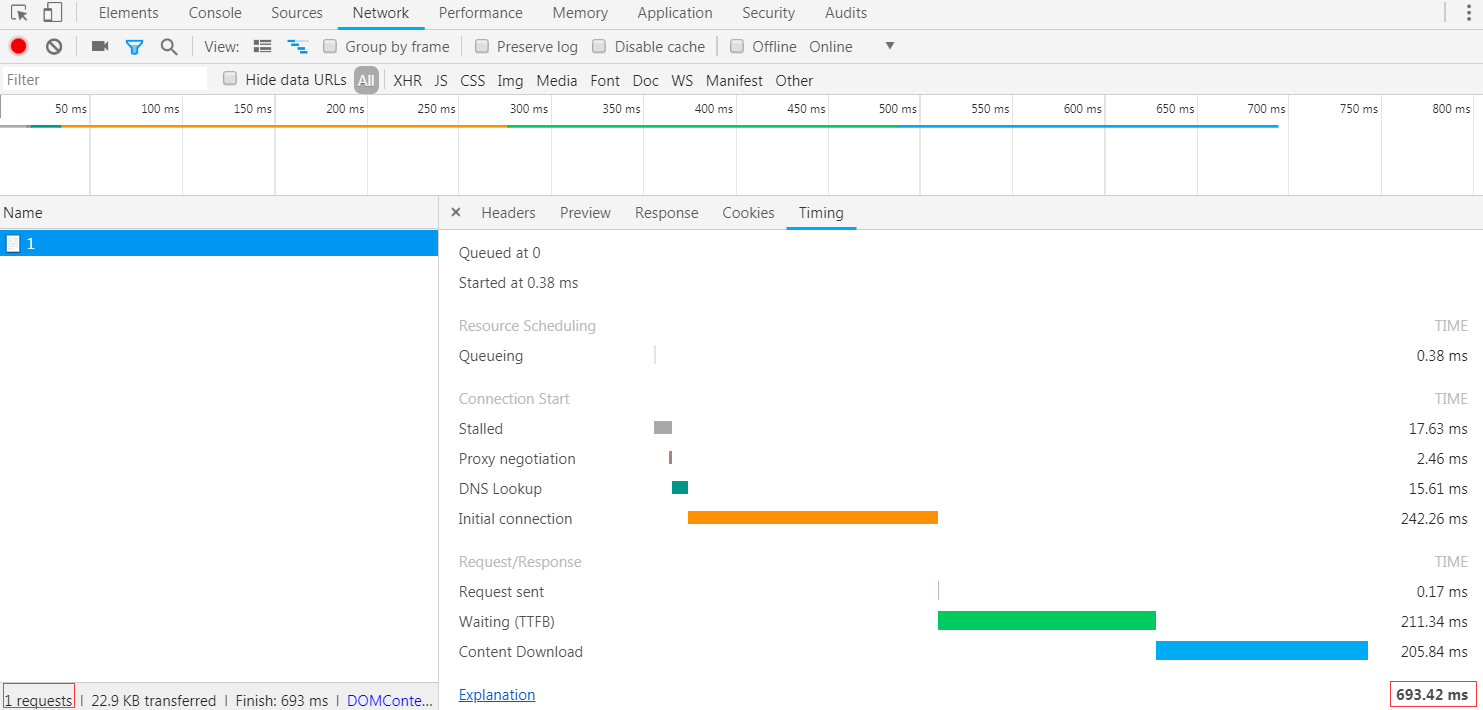
Connecting:TCP建立连接时间

Waiting : Waiting for a response from the server.

Receiving :read entire response from server

Chrome

[Reference](https://developers.google.com/web/tools/chrome-devtools/network-performance/understanding-resource-timing)



Queuing：阻塞的原因

Finish： timeline里所有request总时间

DOMContentLoaded ： 加载html dom内容，不包括js，css等

Load： 整个页面的时间

### 3.5.4 优化策略

#### 7.2.2 线程

系统负载高原因

1.线程切换

理想状态CPU，内存，IO外设都满负荷工作，大部分情况瓶颈都在IO的读写部分，多线程适合计算性case。

CPU主频

CPU内核工作的时钟频率，单位之间产生脉冲个数。用于同步指令执行。数值不代表运算速度(和运算速度相关：cache，指令集 ，CPU位数)，但对提高运算速度至关重要。

cat /boot/config-`uname -r` | grep 'CONFIG\_HZ='

cat /proc/interrupts

中断

TODO 操作系统时间片和中断、主频的关系？

##### 7.2.2.1 线程优化

线程切换开销

[Reference](http://ifeve.com/java-context-switch/)

上下文切换会带来直接和间接两种性能开销

**直接消耗**: CPU寄存器需要保存和加载, 系统调度器的代码需要执行, 刷新TLB(TLB：translation lookaside buffer，快表，可以理解为页表缓冲，地址变换高速缓存), CPU 的pipeline需要刷掉;

**间接消耗**：多核的cache之间得共享数据,对于程序的影响要看线程工作区操作数据的大小。

线程数量

**1) Max Number of threads:**

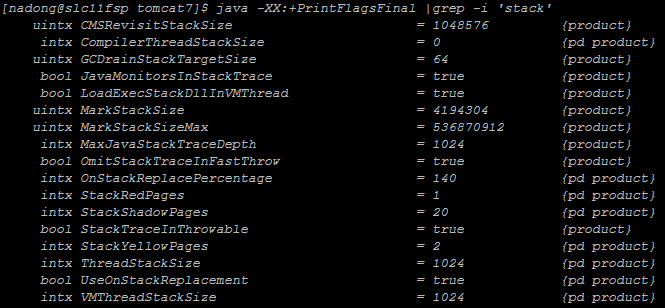
(MaxProcessMemory - JVMMemory - ReservedOsMemory) / ThreadStackSize

* MaxProcessMemory 指的是一个进程的最大内存。在32位的 windows下是 2G
* JVMMemory         JVM内存
* ReservedOsMemory  保留的操作系统内存
* ThreadStackSize      线程栈的大小(-Xss)

2)线程数量

((CUP时间+CUP等待时间)/CUP时间)\*CUP数量

ThreadStackSize



java -XX:+PrintFlagsFinal | grep ThreadStackSize

##### Thread Dump

tomcat

dump分析发现大量WAITING

##### TOMCAT

Tomcat的线程优化，本质上就是java 的线程池优化。结合socket特点来优化。实现线程资源和任务的一种平衡。

Server

超过最大连接数，响应时间变长

最大连接数越大，cup load约高

线程数越大，超过500，响应时间变长。线程切换时间？

吞吐量是否应该在服务端测，排除传输时间干扰。

#### 7.2.2 内存

JVM的编译优化

指令重排序，内存栅栏等

##### 查看内存

PrintFlagsFinal

java -XX:+PrintFlagsFinal

**PrintCommandLineFlags**

已经设置参数

jmap –heap pid

1. Heap Configuration:
2. MinHeapFreeRatio         = 0
3. MaxHeapFreeRatio         = 100
4. MaxHeapSize              = 1073741824 (1024.0MB)
5. NewSize                  = 314572800 (300.0MB)
6. MaxNewSize               = 314572800 (300.0MB)
7. OldSize                  = 759169024 (724.0MB)
8. NewRatio                 = 2
9. SurvivorRatio            = 8
10. MetaspaceSize            = 21807104 (20.796875MB)
11. CompressedClassSpaceSize = 1073741824 (1024.0MB)
12. MaxMetaspaceSize         = 17592186044415 MB
13. G1HeapRegionSize         = 0 (0.0MB)

jstat –gcutil pid 查看内存变化。

##### JAVA OPS优化

**方法区Perm**

* -XX:PermSize=5M -XX:MaxPermSize=7M

**32G 内存配置示例：**

-Xms10g -Xmx10g -Xmn1024m -Xshare:off

**jvm性能调优**

* 控制GC的行为. 频繁GC导致性能下降,因此经常会根据系统运行的程序的特性来更改GC行为
* 调整JVM堆栈大小.xmx xxx
* 控制JVM线程的内存分配.xss

**JVM调试方法：**

1.Jconsole查看内存各部分变化

2.gc 用jstat

##### Heap Dump

jmap -dump:format=b,file=/scratch/tools/heamdump.out 22204

#### 7.2.4 IO

配置文件server.xml，略。

## 4.3 架构设计Case

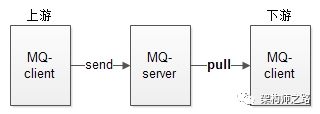
### 秒杀系统

[Reference](https://mp.weixin.qq.com/s?__biz=MzIwMzY1OTU1NQ==&mid=2247484606&idx=1&sn=4f37b694e67747aa16d37dd3ec288225&chksm=96cd44f2a1bacde4d84e02e8027fe77d075d84a689da747d84389c532fb4321255184e662591&mpshare=1&scene=1&srcid=0926E9jrsacrVCbAbdvtioUv#rd])

1）缓冲流量

削峰=缓冲流量。

流量冲击：直接调用/MQ采用push给下游，下游接收方无法控制到达的流量，若调用方不限速，很可能被压垮。



这里下游的MQ-client根据server处理能力pull消息，从队列中处理业务。

2）性能优化

高并发导致程序阻塞**，**用户感受卡顿。

* 做好缓存：Read数据尽量放缓存，切断User和db的直接交互。并提前预热数据。

Warning

从缓存读可能要牺牲强一致性，保证最终一致性。比如从Slave查看，下单操作master

3）控制库存

问题2：库存无法有效控制，出现超卖现象。

分布式锁。

### 设计一个消息中间件

### 消息推送系统

#### Base

[Reference](https://mp.weixin.qq.com/s?__biz=MzI4NDY5Mjc1Mg==&mid=2247485859&idx=1&sn=84e882c025682164d50ab5c8b00c0480&chksm=ebf6d1dcdc8158ca644c874f3701a9d84f01e4ede53c1bb590fa36ebfa346b37b6b68d93bd78&mpshare=1&scene=1&srcid=0929OgZWIOVUIIeKhyZBaU35#rd])

* 技术选型：NIO 1.大量连接 2.双工通信 （选择不多Netty，社区，资料维护)
* 协议解析：定制私有协议。要精简，减少不必要的传输。
* 通道关系：client和Channel的关系维护。单机版用一个map保存即可。分布式环境client可能connect到不同server

#### 实现功能点

上一章节已结束分布式的基础功能，消息推送系统在分布式的基础上还要实现下列

* 消息流转：客户端多，上行量大，不适合在server中，可用消息中间件解耦，比如kafka。
* 心跳：server和client都要心跳，通过注册中心检测各节点的状态

中间件

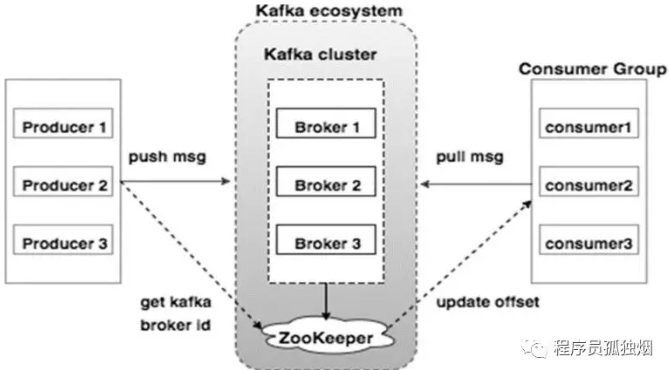
[Reference](https://kb.cnblogs.com/page/196448/)

一般情况，中间件应用于分布式环境，主要解决异构网络环境下模块的互连与互操作问题，提供标准接口、协议，屏蔽实现细节（将具体业务和底层逻辑解耦的组件），提高应用系统移植性。

分类：数据访问中间件，远程调用中间件，消息中间件，交易中间件，对象中间件。

#### 消息队列

kafka：高吞吐的分布式发布订阅消息系统。



# Part 3 软件工程

### 软件类型

这个世界变化实在是快，各种新名词层出不穷，让人眼花缭乱（云，大数据，AR,AI）。其实很多东西只不过新瓶装老酒而已。

1）单机类型

最开始的那些不需要互联网的单机软件。

2）C/S

通过安装的客户端访问服务。早期多为局域网。



3）B/S

传统上软件开发主要针对单机环境,互联网的兴起，越来越多的人开始意识到，网站即软件，而且是一种新型的软件。浏览器/服务器架构。软件云端部署，用户通过通过浏览器来使用。



4）云计算

**服务模式**

SaaS（软件即服务）、PaaS（平台即服务）和IaaS（基础设施即服务）



SaaS

软件即服务，SaaS不是一种软件架构，而是一种软件销售方式。本质上还是采用B/S架构。

但通过对每个使用者收取年租费或月租费来销售

优势：云端办公和较低资费。

目标用户：中小企业用户，因为他们要求不高，不像大企业那般苛刻，通用版开发难度也小。

ERP型SaaS

**部署方式**

私有云、社区云、公有云和混合云

**基本特征**

按需自助服务：按需使用，按需付费

广泛的网络访问、资源共享、快速的可伸缩性和可度量的服务。

企业级云

### 企业级软件

面向的用户为企业，政府，业务比较复杂。按功能划分为财务会计、ERP（企业资源规划）、CRM（客户关系管理）、SCM（供应链管理）、HRM（人力资源管理）、BI（商务智能）、CMS（内容管理系统）和企业通信工具等。也可以按行业划分为制造业、零售业、医药业等解决方案。

ERP

Enterprise Resource Planning，是一种供应链管理思想。包括采购、销售、制造、财务等功能。

#### 复杂度高

政府学校和企业的采购模式、利益诉求不尽相同；不同生产、管理环节对软件特性、服务模式等的需求大相径庭；不同企业对同种软件也总会有个性化的要求；同一个软件在同一个企业内的同一个功能，也可能因部门间的厉害纠葛，导致需求不可理喻地分裂。

#### 国内现状

### 互联网公司

广义上来讲，企业规划、业务的开展都是基于互联网的前提之下施行的，都可以叫互联网公司。企业直接面向真实用户，即TO C。

特点：

追求高效，扁平化管理，去层级化，人本主义。

TODO3：和传统企业有什么本质区别？

2B VS 2C

2C，用户即购买者，更注重用户体验。2B，购买者为决策层，更注重提供生产和管理

国内现状：

用户有限，一天就是 24 小时，除去睡觉、工作，留给互联网的时间非常少，却有成千上万的产品在争夺这少得可怜的时间。而 B 端用户在工作时间使用产品，粘性大，高频刚需，还有可观的服务费收入。

### 开发模式

瀑布模型

将功能的实现与设计分开，将软件分为六个基本活动，并且规定了它们自上而下、相互衔接的固定次序，如同瀑布流水，逐级下落

生命周期：制定计划、需求分析、软件设计、程序编写、软件测试和运行维护

缺点：项目各个阶段之间极少有反馈，不适应用户需求的变化

敏捷开发

敏捷同样需要大量的需求分析，改进瀑布模型，把产品开发引向了小步快速迭代，周期交付。

缺点：不用将业务方方面面考虑周到即可开发，业务模型的频繁变更带来更高的维护成本。

DDD(领域驱动开发)

Domain Driven Design ,DDD is about designing software based on models of the underlying domain” — Martin Fowler

### GPL协议

## 4.5 Design Tools

### 3.6.1 ER图

TODO4

### 3.6.2 UML图

* Implements：接口实现
* Extentd：继承
* Dependence ：依赖。比如方法传递或局部直接new。虚线箭头- - - -🡪
* Association：关联，类中包含另一个类。实线箭头🡪。
* Association根据生命周期分为Composition和aggregation

依赖和关联都是使用到另一个类，关联是一种更强的依赖关系。

1）implements

接口是对类的行为抽象，抽象程度最高



2）Extend



3）Dependence

* 局部new
* 方法的参数传递
* 静态方法的调用;



**association**是把使用到的**类作为成员变量**，分为组合/聚合



Association

关联分为组合/聚合，区别在于存在的生命周期不同。

**聚合(aggregation):**



**组合(Composition)**

mouth是Persion组成部分，有相同的生命周期。 较聚合是一种更强的依赖关系。

1. **public** **class** Person {
2. **private** Mouth mouth;
3. **public** Person() {
4. mouth = **new** Mouth();
5. }
6. }

