学习笔记

Java

目录

Part I

Java 基础知识

第 1 章　基本概念

1.1　基础知识

官方文档

Java Documentation

1.1.1　常用术语

JDK: Java Development Kit

　编写 java 程序的程序员使用的软件

JRE: Java Runtime Environment

　运行 java 程序的用户使用的软件

Server JRE:

　在服务器上运行 java 程序的软件

SE: Standard Edition

　用于桌面或者简单服务器应用的 java 平台

EE: Enterprise Edition

　用于复杂服务器应用的 java 平台

ME: Micro Edition

　用于手机和其他小型设备的 java 平台

Java FX:

　用于图形化用户界面的一个替代工具包在 Oracle 的 Java SE 发布版本中提供

OpenJDK:

　Java SE 的一个免费开源实现，不包含浏览器集成或者 JavaFX

JCP: Java Community Process

　Java 社区进程

JSR: Java Specification Requests

　Java 规范请求由 JCP 成员向委员会提交的 Java 发展议案，经过一系列流程后，如果通过最终会体现在未来的 Java 中

TCK: Technology Compatibility Kit

　技术兼容性测试

1.1.2 Java 版本

Java SE 是相对于 Java EE 和 Java ME，它是 Java 的标准版。

Java SE8 对应“内部”版本号是 1.8.0。

版本号命名规则示例：

Java SE 8u31

Java SE 8 的第 31 次更新，内部版本号为 1.8.0_31

1.1.3 JDK 安装

Mac 可以通过 dmg 文件包安装

安装后相关路径

```
/Library/Java/JavaVirtualMachines/jdk-*.jdk/Contents/Home/

/usr/libexec/java_home

/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/

# 执行目录
/usr/bin/java

# 库源文件为压缩文件src.zip
/Library/Java/JavaVirtualMachines/jdk-*.jdk/Contents/Home/lib/src.zip
```

src.zip 文件包含了所有的公共类的源代码。

设置环境变量

设置 JAVA_HOME

```
        export JAVA_HOME="$(/usr/libexec/java_home -v 1.8.0_201)"       # 以版本1.8.0_201为例
        export PATH=$JAVA_HOME/bin:$PATH
```

1.2 Java Tools

Create and Build Applications

* **javac**

  You can use the javac tool and its options to read Java class and interface definitions and compile them into bytecode and class files.

  您可以使用 javac 工具及其选项来读取 Java 类和接口定义，并将它们编译成字节码和类文件。

  具体使用可参照 7.1 javac 章节

* **javap**

  You use the javap command to disassemble one or more class files.

  您可以使用 javap 命令来反汇编一个或多个类文件。

具体使用可参照 7.2 javap 章节

* **java**

You can use the java command to launch a Java application.

您可以使用 java 命令启动 java 应用程序。

具体使用可参照 7.3 java 章节

* **jar**

You can use the jar command to create an archive for classes and resources, and to manipulate or restore individual classes or resources from an archive.

您可以使用 jar 命令为类和资源创建存档，并从存档中操作或还原单个类或资源。

* **jdeps**

You use the jdeps command to launch the Java class dependency analyzer.

使用 jdeps 命令启动 Java 类依赖关系分析器。

* **javadoc**

You use the javadoc tool and its options to generate HTML pages of API documentation from Java source files.

您可以使用 javadoc 工具及其选项从 Java 源文件生成 API 文档的 HTML 页面。

Monitor Java Applications

* **jconsole**

You use the jconsole command to start a graphical console to monitor and manage Java applications.

您可以使用 jconsole 命令启动图形化控制台来监视和管理 Java 应用程序。

* **jvisualvm**

Monitor the JVM

* **jps**

Experimental You use the jps command to list the instrumented JVMs on the target system.

实验性。您可以使用 jps 命令列出目标系统上已检测的 jvm。

具体使用可参照 7.4 jps 章节

* **jstat**

Experimental You use the jstat command to monitor JVM statistics.

实验性。使用 jstat 命令监视 JVM 统计数据。此命令是实验性的，不受支持。

* **jstatd**

Experimental You use the jstatd command to monitor the creation and termination of instrumented Java HotSpot VMs.

实验性。您可以使用 jstatd 命令监视仪表化 Java HotSpot vm 的创建和终止。

* **jmc**

  You use the jmc command and its options to launch Java Mission Control. Java Mission Control is a profiling, monitoring, and diagnostics tools suite.

  使用 jmc 命令及其选项启动 Java 任务控制。Java 任务控制是一个分析、监视和诊断工具套件。

Troubleshooting Tools

* **jcmd**

  You use the jcmd utility to send diagnostic command requests to a running Java Virtual Machine (JVM).

* **jdb**

  You use the jdb command and its options to find and fix bugs in Java platform programs.

* **jinfo**

  Experimental You use the jinfo command to generate Java configuration information for a specified Java process.

* **jmap**

  Experimental You use the jmap command to print details of a specified process.

  实验性。您可以使用 jmap 命令打印指定进程的详细信息。

* **jstack**

  Experimental You use the jstack command to print Java stack traces of Java threads for a specified Java process.

  实验性。您可以使用 jstack 命令为指定的 Java 进程打印 Java 线程的 Java 堆栈跟踪。

Others

* **jlink**

  You can use the jlink tool to assemble and optimize a set of modules and their dependencies into a custom runtime image.

  您可以使用 jlink 工具将一组模块及其依赖项组装并优化到自定义运行时映像中。

* **jmod**

  You use the jmod tool to create JMOD files and list the content of existing JMOD files.

  使用 jmod 工具创建 JMOD 文件并列出现有 JMOD 文件的内容。

* **jdeprscan**

  You use the jdeprscan tool as a static analysis tool that scans a jar file (or some other aggregation of class files) for uses of deprecated API elements.

  您可以使用 jdeprscan 工具作为静态分析工具，它扫描一个 jar 文件（或一些其他类文件的聚合），以便使用废弃的 API 元素。

* **jshell**

You use the jshell tool to interactively evaluate declarations, statements, and expressions of the Java programming language in a read-eval-print loop (REPL).

您可以使用 jshell 工具在 read-eval-print 循环（REPL）中交互式地计算 Java 编程语言的声明、语句和表达式。

更详细的内容请参考
tools-and-command-reference

1.3 规范

Java 区分大小写

Java 采用骆驼命名法，CamelCase。

数据类型

Java 包含 8 种基本类型。

- 4 种整型
- 2 种浮点类型
- 1 种表示 Unicode 编码的字符单元的字符类型 char
- 1 种表示真值的 boolean 类型

整型

类型	存储需求	取值范围
byte	1 字节	-128 ~ 127
short	2 字节	-32768 ~ 32767
int	4 字节	-2147483648 ~ 2147483647
long	8 字节	-9223372036854775808 ~ 9223372036854775807

Java 没有任何无符号（unsigned）形式的整型。

长整型值有一个后缀 l 或者 L。

```
        long number = 123L;             // number值为: 123
```

十六进制带前缀 0x 或者 0X

```
        int number = 0x123;
```

八进制带前缀 0

从 Java7:

* 加上前缀 0b 或者 0B 就可以写二进制数。

```
        int number = 0b100;             // number值为：4
```

* 为数字加下划线，Java 编译器会去除这些下划线

```
        Sytem.out.println(0_1_0);       // 010表示八进制，输出8
```

注意 JS INT 类型的值范围：

- Integers in JavaScript
- Safe Integers

浮点数

类型	存储需求	取值范围
float	4 字节	有效位数为 6-7 位
double	8 字节	有效位数为 15 位

float 类型的值要有后缀 f 或者 F。没有后缀默认为 double 类型。

double 类型也可以添加 d 或者 D 后缀。

用于表示溢出或者出错情况的三个特殊浮点值：

- 正无穷大
- 负无穷大
- NaN （不是一个数字）

例如： 一个正整数除以 0 的结果为正无穷大；计算 0/0 或者负数的平方根结果为 NaN。

对应常量为：

```
        // float
        Float.POSITIVE_INFINITY;
        Float.NEGATIVE_INFINITY;
        Float.NaN;
        // double
        Double.POSITIVE_INFINITY;
        Double.NEGATIVE_INFINITY;
        Double.NaN;
```

所有“非数值”的值都认为是不相同的。

```
        if(Double.isNaN(x))   // check whether x is "not a number"
```

char 类型

常量

关键字 final

类常量关键字 `static final`

ava 类中各成员初始化顺序

基类静态代码块、基类静态成员字段并列优先级，按照代码中出现先后顺序执行（只有第一次加载类时执行）

派生类静态代码块、派生类静态成员字段并列优先级，按照代码中出现先后顺序执行（只有第一次加载类时执行）

基类普通代码块、基类普通成员字段并列优先级，按照代码中出现先后顺序执行

基类构造函数

派生类普通代码块、派生类普通成员字段并列优先级，按照代码中出现顺序执行

派生类构造函数

switch 语句

Java

只支持 (byte,short,char,int) 基本类型,Enum 类型以及 String,Character,Byte,Short 和 Integer 类。

只支持常量表达式。

case 会进行整数范围验证。

```
    /*
    * 枚举类型
    */
    public enum Day {
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
        THURSDAY, FRIDAY, SATURDAY
    }

    /*
    * 星期判断
    * @param day
    */
    public static void switchTestWithEnum(Day day) {
    switch (day) {
        case MONDAY:
            System.out.println("Today is Mondays.");
            break;
        default:
            System.out.println("Today is ...");
            break;
        }
   }

    public static void switchTestWithCharacter(Character character) {
        switch (character) {
            case 'a':
                System.out.println("character is a");
                break;
            case 'b':
                System.out.println("character is b");
            default:
                System.out.println("nothing...");
        }
    }

    //调用
    switchTestWithEnum(Day.SUNDAY);
    switchTestWithCharacter(new Character('a'));
```

code snippet 1.1: swtich

PHP

case 表达式可以是任何求值为简单类型的表达式，即整型或浮点数以及字符串。不能用数组或对象，除非它们被解除引用成为简单类型。

允许使用分号代替 case 语句后的冒号

case 比较执行的是松散比较 ==

```
    $beer = 'caseC';
    $a ='caseA';

    switch($beer)
    {
        case $a;
            echo "this is caseA";
        break;
        case 'caseB';
            echo "this is caseB";
        break;
```

```
        case 'case'.'C':
            echo "this is caseC";
        break;
        default;
            echo 'Please make a new selection...';
        break;
    }
```

第 2 章　字符串

Strings are constant; their values cannot be changed after they are created.

2.1　字符串对象

String 对象是不可改变的，具有恒定性。

每个 String 对象都有常量值。

字符串字面常量是对 String 类的示例的饮用。

在 Java 中 String 对象可以认为是 char 数组的延伸和进一步的封装。

这里所说的 char 数组不是 C 语言意义上的字符型数组，而是大致类似于 C 语言中的 char* 指针。

```
String str = "Java学习笔记";
```

```
char* str = "Java学习笔记";
```

```
public final class String implements java.io.Serializable, Comparable<String>, CharSequence {

/** The value is used for character storage. */
private final char value[];

/*
* @param  value             Array that is the source of characters
* @param  offset            The initial offset
* @param  count             The length
*
* @throws  IndexOutOfBoundsException
*          If the {@code offset} and {@code count} arguments index
*          characters outside the bounds of the {@code value} array
*/
public String(char value[], int offset, int count) {
    if (offset < 0) {
        throw new StringIndexOutOfBoundsException(offset);
    }
    if (count <= 0) {
        if (count < 0) {
            throw new StringIndexOutOfBoundsException(count);
        }
        if (offset <= value.length) {
            this.value = "".value;
            return;
        }
    }
    // Note: offset or count might be near -1>>>1.
    if (offset > value.length - count) {
        throw new StringIndexOutOfBoundsException(offset + count);
    }
    this.value = Arrays.copyOfRange(value, offset, offset+count);
```

```
    }

    }
```

通过上面 String 类的实现代码可以发现 String 类和 value 数组都是 final 类型，这就保证了 String 对象的不变性。这种不变性可以带来极大的好处。

- 保证对 String 对象的任意操作都不会改变原字符串
- 意味着操作字符串不会出现线程同步问题
- 成就了字符串驻留以及共享（常量池）

2.2 字符串连接操作符 +

如果字符串连接操作符运算的结果不是编译时的常量表达式，那么该操作符会隐式地创建新的 String 对象。

如果只有一个操作数表达式是 String 类型，那么就会在另一个操作数上执行字符串转换以在运行时产生字符串。对于简单类型 Java 还可以通过直接将简单类型转换为字符串而优化掉包装器对象的创建。

+操作字符在语法上是左结合。

例如：

```
    String first = 1 + 2 + "Java";
    String second = "Java" + 1 + 2;
    String three = "Java" + null;
    System.out.println(first);
    System.out.println(second);
    System.out.println(three);

    /**
    * Output:
    *
    * 3Java
    * Java12
    * Javanull
    *
    */
```

在字符串转换方面要特别注意引用类型转行。

如果该引用是 null，那么它转换为字符串"null"。

如果其他引用类型，则调用其对象上的 toString() 方法；如果调用 toString() 方法的结果是 null，那么就用字符串"null" 代替。

第 3 章　对象

3.1　初始化

静态初始化只有在对象被创建或者第一次访问静态数据时才会被初始化。

初始化的顺序时先静态对象（如果他们尚未因前面的对象创建过程而被初始化），而后才是“非静态”对象。

以 Dog 类为示例总结一下对象的创建过程：

1. 当首次创建类型为 Dog 的对象或者 Dog 类的静态方法/静态域首次被访问时，java 解释器必须查找类路径，以定位 Dog.class 文件

2. 然后载入 Dog.class ，有关静态初始化的所有动作都会执行。因此，静态初始化只在 Class 对象首次加载的时候进行一次。按照顺序。

3. 当用 new Dog() 创建对象的时候，首先将在堆上为 Dog 对象分配足够的存储空间。

4. 这块存储空间会被清零，这就自动将 Dog 对象中的所有基本类型数据都设置成了默认值（数字为 0，boolean 为 false），而引用则被设置成 null（例如 String）

5. 执行所有出现于字段定义处的初始化动作

6. 执行构造器（会涉及到继承的问题）

总结

基类静态代码块、基类静态成员字段并列优先级，按照代码中出现先后顺序执行（只有第一次加载类时执行）

派生类静态代码块、派生类静态成员字段并列优先级，按照代码中出现先后顺序执行（只有第一次加载类时执行）

基类普通代码块、基类普通成员字段并列优先级，按照代码中出现先后顺序执行

基类构造函数

派生类普通代码块、派生类普通成员字段并列优先级，按照代码中出现顺序执行

派生类构造函数

3.1.1　对象拷贝

对象的拷贝分为 shallow copy 和 deep copy 。

shallow copy 既浅拷贝。

https://zhuanlan.zhihu.com/p/26964202

clone

clone

PHP 与 Java 一样，对象的对象属性都只是引用拷贝.

php 的拷贝是通过 clone 关键字实现。

第4章　范型

4.1　基本概念

范型实现了参数化类型的概念。

4.2　范型类

4.3　范型方法

4.4　通配符类型

函数式接口

关于函数式接口

如果一个接口只有一个抽象方法，那么该接口就是一个函数式接口

如果我们在某个接口上声明了 FunctionalInterface 注解，那么编译器就会按照函数式接口的定义来要求该接口

如果某个接口只有一个抽象方法，但我们并没有给该接口声明 FunctionalInterface 注解，那么编译器依旧会将该接口看作是函数式接口

```
    List<Integer> list = Arrays.asList(1,2,3,4,5);

    list.forEach(new Consumer<Integer>(){
        @Override
        public void accept(Integer integer){
            System.out.println(integer);
        }
    });
```

在 java 中 Lambda 表达式是对象，他们必须依附与一类特别的对象类型-函数式接口（functional interface)

That instances of functional interfaces can be created with lambda expressions, method references, or constructor references.

函数式接口的实例可以通过 lambda 表达式，方法引用和构造函数引用创建。

外部迭代和内部迭代

Java lambda 表达式是一种匿名函数；它是没有声明的方法，即没有访问修饰符，返回值声明和名字。

lambda 操作符：-> lambda 左边：接口中抽象方法的形参列表 lamdba 右边：重写抽象方法的方法体

当只有一个参数，并且类型可推导时，圆括号 () 可省略。例如： a-> return a*a

lambda 表达式的主体可包含零条或多条语句

如果 lambda 表达式的主体只有一条语句，花括号 可省略。匿名函数的返回类型与该主体表达式一致

如果 Lambda 表达式的主体包含一条以上语句，则表达式必须包含在花括号中。匿名函数的返回类型与代码块的返回类型一致，若没有返回值则为空

Function<T, R>

BiFunction<T, U, R>

第 5 章　异常处理

5.1　异常

异常的抛出同其他对象的创建一样，将使用 new 在堆上创建异常对象。然后当前的执行路径（它不能继续执行下去）被终止，并且从当前环境中弹出对异常对象的引用。此时异常处理机制接管程序，并开始寻找一个恰当的地方来继续执行程序。而这个恰当的地方就是“异常处理程序”。它的任务是将程序从错误状态中恢复，以使程序能要么换一种方式运行，要么继续运行下去。

Throwable 类是所有异常的基类。

//此处有类图

Throwable Error Exception RuntimeException 非 RuntimeException

5.2　异常说明

异常说明使用附加关键字 throws。

重抛异常会把异常抛给上一级环境中的异常处理程序。同一个 try 块的后续 catch 子句将被忽略。

```
/**
 * The {@code Throwable} class is the superclass of all errors and
 * exceptions in the Java language. Only objects that are instances of this
 * class (or one of its subclasses) are thrown by the Java Virtual Machine or
 * can be thrown by the Java {@code throw} statement. Similarly, only
 * this class or one of its subclasses can be the argument type in a
 * {@code catch} clause.
 *
 * For the purposes of compile-time checking of exceptions, {@code
 * Throwable} and any subclass of {@code Throwable} that is not also a
 * subclass of either {@link RuntimeException} or {@link Error} are
 * regarded as checked exceptions.
 *
 * <p>Instances of two subclasses, {@link java.lang.Error} and
 * {@link java.lang.Exception}, are conventionally used to indicate
 * that exceptional situations have occurred. Typically, these instances
 * are freshly created in the context of the exceptional situation so
 * as to include relevant information (such as stack trace data).
 *
 * <p>A throwable contains a snapshot of the execution stack of its
 * thread at the time it was created. It can also contain a message
 * string that gives more information about the error. Over time, a
 * throwable can {@linkplain Throwable#addSuppressed suppress} other
 * throwables from being propagated.  Finally, the throwable can also
 * contain a <i>cause</i>: another throwable that caused this
 * throwable to be constructed.  The recording of this causal information
 * is referred to as the <i>chained exception</i> facility, as the
 * cause can, itself, have a cause, and so on, leading to a "chain" of
 * exceptions, each caused by another.
 *
 * <p>One reason that a throwable may have a cause is that the class that
 * throws it is built atop a lower layered abstraction, and an operation on
 * the upper layer fails due to a failure in the lower layer.  It would be bad
 * design to let the throwable thrown by the lower layer propagate outward, as
 * it is generally unrelated to the abstraction provided by the upper layer.
 * Further, doing so would tie the API of the upper layer to the details of
 * its implementation, assuming the lower layer's exception was a checked
 * exception.  Throwing a "wrapped exception" (i.e., an exception containing a
 * cause) allows the upper layer to communicate the details of the failure to
 * its caller without incurring either of these shortcomings.  It preserves
 * the flexibility to change the implementation of the upper layer without
```

```
 * changing its API (in particular, the set of exceptions thrown by its
 * methods).
 *
 * <p>A second reason that a throwable may have a cause is that the method
 * that throws it must conform to a general-purpose interface that does not
 * permit the method to throw the cause directly.  For example, suppose
 * a persistent collection conforms to the {@link java.util.Collection
 * Collection} interface, and that its persistence is implemented atop
 * {@code java.io}.  Suppose the internals of the {@code add} method
 * can throw an {@link java.io.IOException IOException}.  The implementation
 * can communicate the details of the {@code IOException} to its caller
 * while conforming to the {@code Collection} interface by wrapping the
 * {@code IOException} in an appropriate unchecked exception.  (The
 * specification for the persistent collection should indicate that it is
 * capable of throwing such exceptions.)
 *
 * <p>A cause can be associated with a throwable in two ways: via a
 * constructor that takes the cause as an argument, or via the
 * {@link #initCause(Throwable)} method.  New throwable classes that
 * wish to allow causes to be associated with them should provide constructors
 * that take a cause and delegate (perhaps indirectly) to one of the
 * {@code Throwable} constructors that takes a cause.
 *
 * Because the {@code initCause} method is public, it allows a cause to be
 * associated with any throwable, even a "legacy throwable" whose
 * implementation predates the addition of the exception chaining mechanism to
 * {@code Throwable}.
 *
 * <p>By convention, class {@code Throwable} and its subclasses have two
 * constructors, one that takes no arguments and one that takes a
 * {@code String} argument that can be used to produce a detail message.
 * Further, those subclasses that might likely have a cause associated with
 * them should have two more constructors, one that takes a
 * {@code Throwable} (the cause), and one that takes a
 * {@code String} (the detail message) and a {@code Throwable} (the
 * cause).
 *
 * @author  unascribed
 * @author  Josh Bloch (Added exception chaining and programmatic access to
 *          stack trace in 1.4.)
 * @jls 11.2 Compile-Time Checking of Exceptions
 * @since JDK1.0
 */
```

第 6 章　线程

6.1　线程基础知识

6.1.1　线程状态

JVM 线程的状态定义在 Thread.State 枚举中。包括:

- NEW
- RUNNABLE
- BLOCKED
- WAITING
- TIMED_WAITING
- TERMINATED

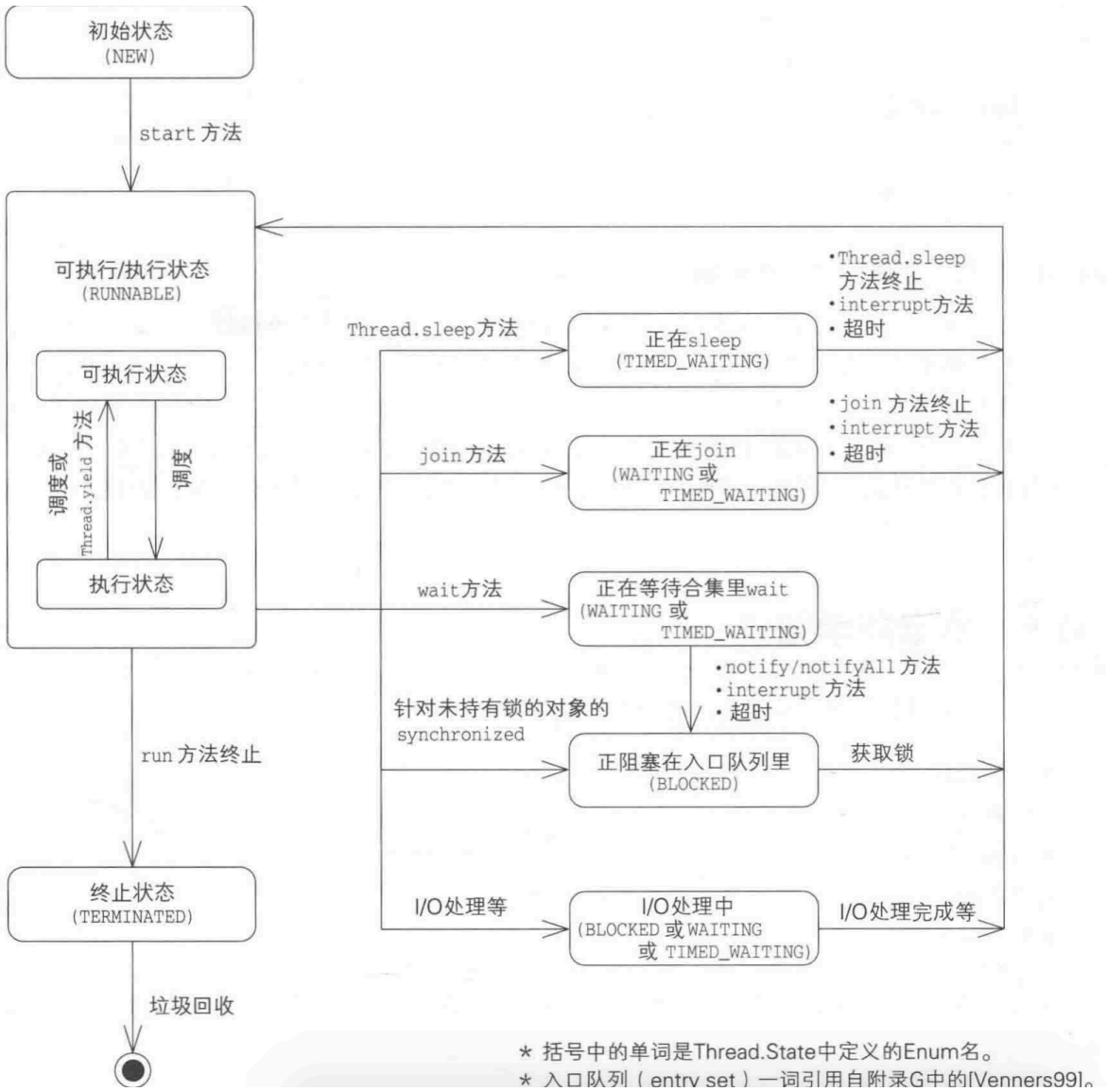


图 6.1: 线程状态迁移图 (摘自《图解 Java 多线程设计模式》)

下面为 `State` 枚举

```
public enum State {
        /**
         * 尚未启动的线程的线程状态
         */
        NEW,

        /**
         * 可运行线程的线程状态。 处于可运行状态的线程正在Java虚拟机中执行，
         * 但是它可能正在等待来自操作系统（例如处理器）的其他资源。
         */
        RUNNABLE,

        /**
         * 线程的阻塞状态，等待监视器锁定。
         * 处于阻塞状态的线程正在等待一个监视器锁进入一个同步块/方法，
         * 或者在调用Object.wait()后重新进入一个同步块/方法。
         */
        BLOCKED,

        /**
         * 等待线程状态
         * 一个线程由于调用下列方法之一而处于等待状态：
         *
         *   Object.wait() 未超时
```

```
         *   Thread.join() 未超时
         *   LockSupport.park()
         *
         * <p>处于等待状态的线程正在等待另一个线程执行特定的操作。
         * 例如: 一个线程在一个对象上已经调用了 Object.wait() ，并正在等待在另一个线程在另一对象上调用
         * Object.notify() or Object.notifyAll()。
         * 调用 thread.join() 的线程正在等待指定的线程终止。
         */
        WAITING,

        /**
         * 具有指定等待时间的等待线程的线程状态。
         * 线程处于定时等待状态，因为调用以下方法之一与指定的正等待时间：
         *
         *   Thread.sleep()
         *   Object.wait(long)带超时参数
         *   Thread.join(long)带超时参数
         *   LockSupport.parkNanos
         *   LockSupport.parkUntil
         *
         */
        TIMED_WAITING,

        /**
         * 终止状态。线程已经完成执行。
         */
        TERMINATED;
    }
```

6.1.2 常用函数

6.2 Synchronized

第7章　Java Tools

7.1 javac 命令

Reads Java class and interface definitions and compiles them into bytecode and class files.

7.2 javap 命令

use the javap command to disassemble one or more class files.

```
用法: javap <options> <classes>

  -version                 版本信息
  -v  -verbose             输出附加信息
  -l                       输出行号和本地变量表
  -public                  仅显示公共类和成员
  -protected               显示受保护的/公共类和成员
  -package                 显示程序包/受保护的/公共类和成员 (默认)
  -p  -private             显示所有类和成员
  -c                       对代码进行反汇编
  -s                       输出内部类型签名
  -sysinfo                 显示正在处理的类的系统信息 (路径, 大小, 日期, MD5 散列)
  -constants               显示最终常量
  -classpath <path>        指定查找用户类文件的位置
  -cp <path>               指定查找用户类文件的位置
  -bootclasspath <path>    覆盖引导类文件的位置
```

示例:

```
javap -c Main.class
```

7.2.1 Java 字节码

Instruction set

instructions fall into a number of broad groups:

- 加载和存储指令 (Load and store)(e.g. aload_0, istore)
- 算术与逻辑指令 (Arithmetic and logic) (e.g. ladd, fcmpl)
- 类型转换指令 (Type conversion) (e.g. i2b, d2i)
- 对象创建与操作指令 (Object creation and manipulation) (new, putfield)
- 堆栈操作指令 (Operand stack management) (e.g. swap, dup2)
- 控制转移指令 (Control transfer) (e.g. ifeq, goto)
- 方法调用与返回指令 (Method invocation and return) (e.g. invokespecial, areturn)

大多数的指令有前缀和（或）后缀来表明其操作数的类型。如下表:

Many instructions have prefixes and/or suffixes referring to the types of operands they operate on.

表 7.1: 前缀/后缀类型对照表

Prefix/suffix	Operand type
i	integer
l	long
s	short
b	byte
c	character
f	float
d	double
a	reference

`Java bytecode`

表 7.2: Java 字节码

mnemonic	stack [before]->[after]	description
aload_0	objectref	load a reference onto the stack from local variable 0

`bytecode`

`code`

`jvms`

7.3 java 命令

Launches a Java application.

```
java [options] classname [args]
java [options] -jar filename [args]
```

通过启动 JRE 来调用指定的类，调用此类的 main() 方法。

```
public static void main(String[] args)
```

7.3.1 选项

- 标准选项
- 非标准选项
- 高级运行时选项（Advanced Runtime Options）
- 高级 JIT 编译器选项（Advanced JIT Compiler Options）
- 高级可维护性选项（Advanced Serviceability Options）
- 高级垃圾收集选项（Advanced Garbage Collection Options）

标准选项

Java 虚拟机（JVM）的所有实现都保证支持的标准选项。

* -agentlib:libname[=options]

加载指定的本机代理库。在库名之后，可以使用特定于库的以逗号分隔的选项列表。

如果指定 -agentlib:foo 选项，则 JVM 尝试加载位于由系统环境变量名为 LD_LIBRARY_PATH（在 OS X 系统下变量名为 DYLD_LIBRARY_PATH）指定位置下的 libfoo.so 的库。

下面的示例将展示如何加载堆分析工具 (HPROF) 库，并且获取堆栈深度为 3，每 20msCPU 的简单采样信息：

```
-agentlib:hprof=cpu=samples,interval=20,depth=3
```

下面这个示例将展示如何加载 Java 调试线协议库并且监听 8000 端口的套接字连接，在主类加载之前挂起 JVM：

```
-agentlib:jdwp=transport=dt_socket,server=y,address=8000
```

XX:+PrintStringTableStatistics

7.4 jps 命令

Lists the instrumented Java Virtual Machines (JVMs) on the target system.

只适用于 HotSpot 虚拟机。

```
jps [ options ] [ hostid ]
```

hostid 可以是进程的标识符或者类似于 URL 的 [protocol:][[//]hostname][:port][/servername] 地址

如果 jps 命令没有指定 hostid 参数，那么工具只搜索本机运行的 JVM。如果指定了 hostid，那么他将通过指定的地址来搜索 JVM。使用指定 hostid 的主机必须运行着 jstatd 进程。

7.4.1 选项

-q

只输出 JVM 标识符的列表，不输出类名、JAR 文件名以及传递给 main 方法的参数。

-m

输出传递给 mian 方法的参数。嵌入式 JVMS 可能输出为空。

-l

输出应用程序主类的完整包名或应用程序 JAR 文件的完整路径名。

-v

输出传递给 JVM 的参数。

-V

只输出本地 JVM 标识符，不输出类名、JAR 文件以及传递给 main 方法的参数。

-Joption

将选项传递给 JVM，其中的选项是 Java 应用程序启动程序参考页面中描述的选项之一。例如，-J-Xms48m 将启动内存设置为 48 MB。

输出格式：

```
lvmid [ [ classname | JARfilename | "Unknown"] [ arg* ] [ jvmarg* ] ]
```

示例：

```
jps

    18027 Java2Demo.JAR
    18032 jps
    18005 jstat
```

```
jps -m remote.domain:2002

       3002 /opt/jdk1.7.0/demo/jfc/Java2D/Java2Demo.JAR
       3102 sun.tools.jstatd.jstatd -p 2002
```

7.5 jcmd 命令

Sends diagnostic command requests to a running Java Virtual Machine (JVM).

```
jcmd [-l|-h|-help]

jcmd pid|main-class PerfCounter.print

jcmd pid|main-class -f filename

jcmd pid|main-class command[ arguments]
```

jcmd 工具向 JVM 发送诊断命令请求。必须在当前运行着 JVM 机器上执行，并且具有与 JVM 具有相同的 user 和 group 标识。

运行 jcmd 不带参数或者使用 l 选项时，相当于 jps (7.4)，会输出 java 进程标识符。

如果将进程标识符 (pid) 或主类 (main-class) 作为第一个参数，则 jcmd 将诊断命令请求发送给具有指定标识符或发具有指定 main-class 名称的 Java 进程。

如果使用 0 作为进程标识符，则会将诊断命令请求发送给所有可用的 Java 进程。

* Perfcounter.print

打印指定 Java 进程可用的性能计数器。性能计数器的列表可能因 Java 进程而异。

* -f filename

从文件中读取诊断命令。文件中每个命令必须单独一行。以 # 开头的命令会被忽略。如果命令行中包含stop关键字则结束命令行读取。

* command [arguments]

要发送到指定 Java 进程的命令。可以通过向该进程发送help命令来获得给定进程的可用诊断命令列表。

如果参数中包含空格，则必须使用单引号或者双引号扩起来。注意使用转义字符 (\) 转义单/双引号。

```
jcmd 15 help     # 15为java进程id

 #### output ####

 15:
 The following commands are available:
 JFR.stop
 JFR.start
 JFR.dump
 JFR.check
 VM.native_memory
 VM.check_commercial_features
 VM.unlock_commercial_features
 ManagementAgent.stop
 ManagementAgent.start_local
 ManagementAgent.start
 VM.classloader_stats
 GC.rotate_log
 Thread.print
 GC.class_stats
 GC.class_histogram
 GC.heap_dump
 GC.finalizer_info
 GC.heap_info
 GC.run_finalization
 GC.run
 VM.uptime
 VM.dynlibs
 VM.flags
 VM.system_properties
 VM.command_line
```

```
    VM.version
    help
```

```
    jcmd 15 help GC.heap_info
```

第 8 章　内存模型

8.1　Java 内存模型与线程

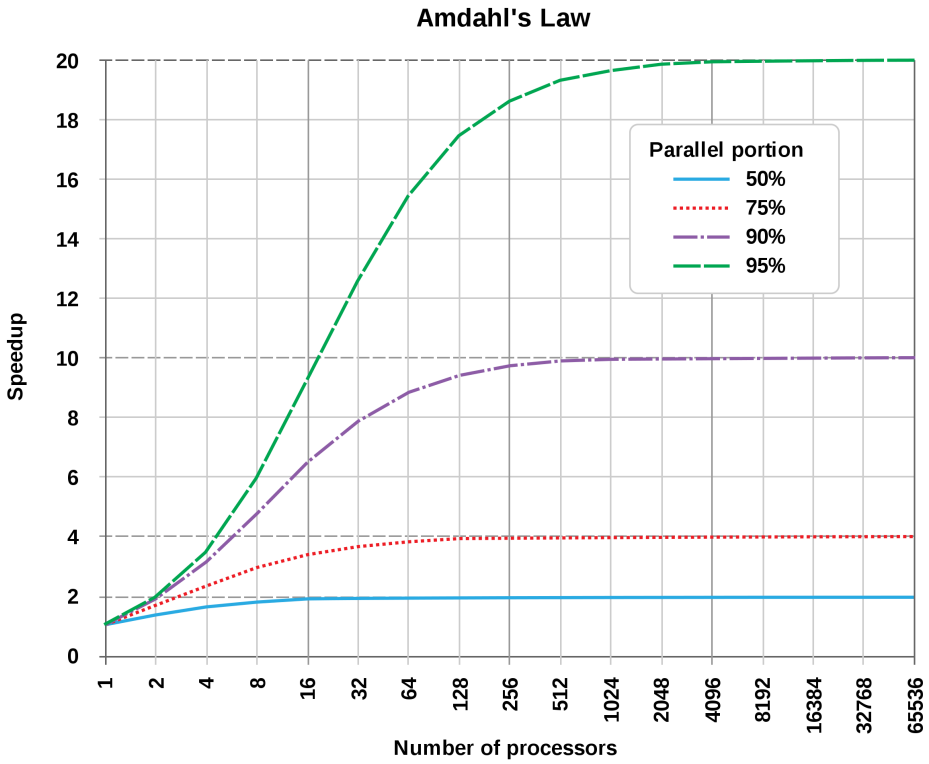


图 8.1: Amdahl's Law 阿姆达尔定律

并发不得不知的阿姆达定律。

一个程序（或者一个算法）可以按照是否可以被并行化分为下面两个部分：

- 可以被并行化的部分
- 不可以被并行化的部分

程序串行执行的总时间我们记为 **T**。

时间 **T** 包括：不可以被并行和可以被并行部分的时间。不可以并行的部分记为 **B**，可以被并行的部分就是 **T – B**。即 **T = B + (T –B)**

定义如下：

```
T = 串行执行的总时间
B = 不可以并行的总时间
T – B = 并行部分的总时间
```

T – B 是可并行化的部分，以并行的方式执行可以提高程序的执行速度。可以提速多少取决于有多少线程或者

多少个 CPU 来执行。线程或者 CPU 的个数我们记为 **N**。可并行化部分被执行的最快时间可以通过下面的公式计算出来：

(T –B) / N 或 **(1/N) * (T – B)**

根据阿姆达定律，当可并行部分使用 `N`个线程或 CPU 执行时，程序的总执行时间为**:**

T(N) = B + (T – B) / N

T(N) 表示并行因子为 N 的总执行时间。T(1) 就是并行因子为 1 时的总执行时间。T(N) = B + (T(1) –B) / N

示例

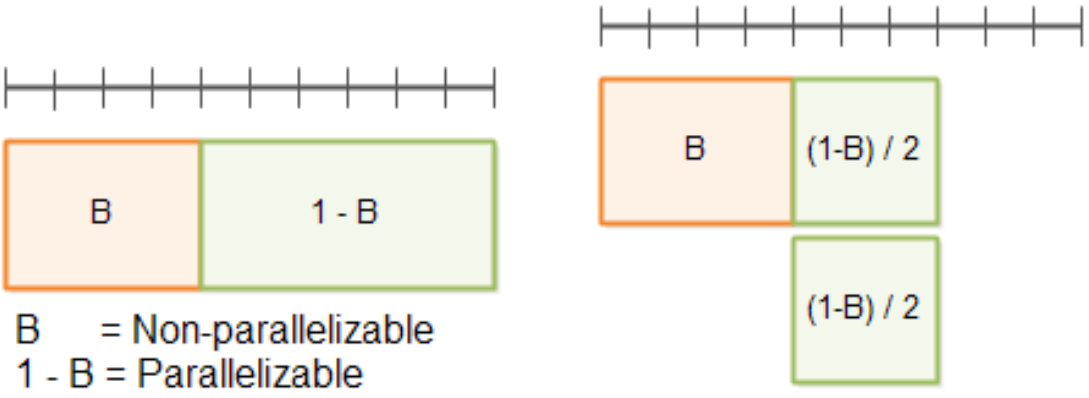
```
/*
 * 总时间为1。串行时间为0.4，那么并行时间为0.6。
 */

// 并行因子为2时
T(2) = 0.4 + ( 1 – 0.4 ) / 2
 = 0.4 + 0.6 / 2
 = 0.4 + 0.3
 = 0.7

// 并行因子为5时
 T(5) = 0.4 + ( 1 – 0.4 ) / 5
 = 0.4 + 0.6 / 6
 = 0.4 + 0.12
 = 0.52
```

图示

并行因子分别为 1,2,3 时。



(a) 并行因子 =1 (b) 并行因子 =2

图 8.2: 并行因子分别为 1，2 时

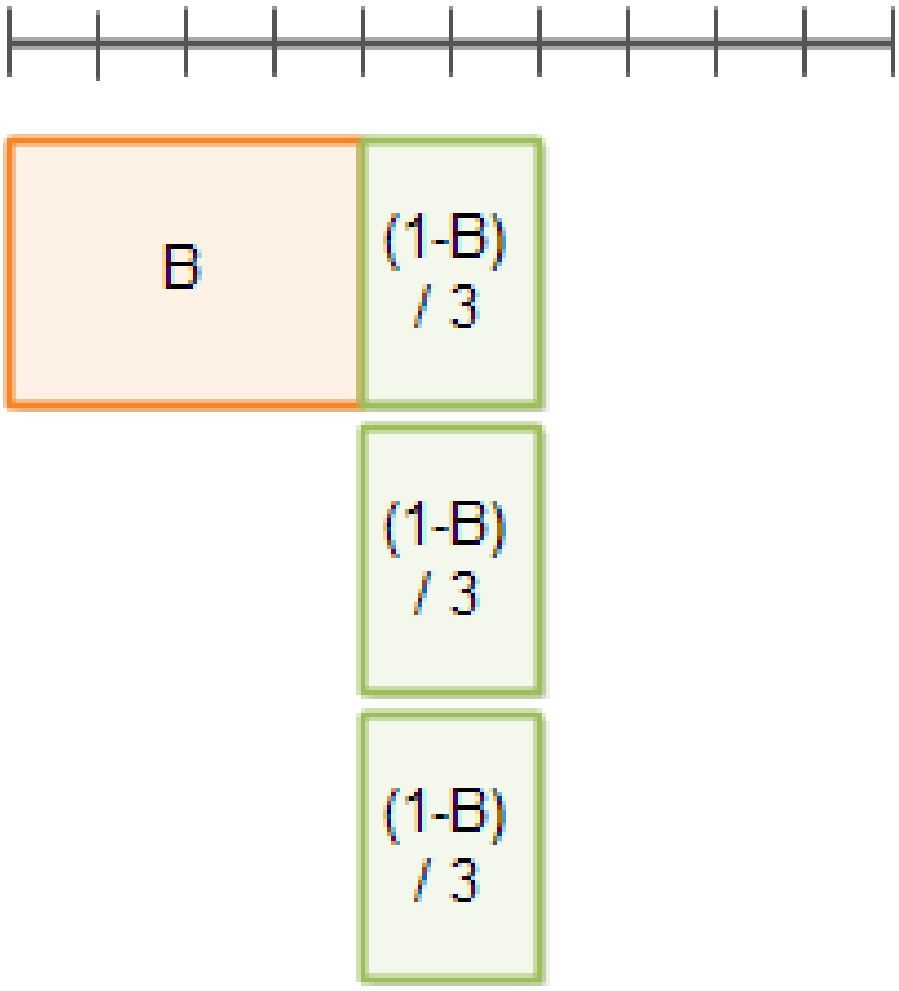


图 8.3: 并行因子 =3

定义

并行计算中的**加速比**是用并行前的执行速度和并行后的执行速度之比来表示的，它表示了在并行化之后的效率提升情况。

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

图 8.4: 阿姆达尔定律

`S(latency)` 代表理论上的加速比

s 为并行处理结点个数

p 为并行计算部分所占比例

1−p 为串行计算部分所占比例

这样:

当 p = 1 时，最大加速比 p = s,

当 p = 0 时，最小加速比 S = 1,

当 s→∞ 时，极限加速比 S→ 1 / (1−p)，这也就是加速比的上限。

例如，若加速前并行代码执行时间占整个代码的执行时间的 75%(p=0.75)，则加速后并行处理的总体性能的提升不可能超过原先的 4 倍。

因此可以推断出:

$$\begin{cases} S_{\text{latency}}(s) \leq \frac{1}{1-p} \\ \lim_{s\to\infty} S_{\text{latency}}(s) = \frac{1}{1-p}. \end{cases}$$

图 8.5: 阿姆达尔定律

阿姆达尔定律强调：当串行换比例一定时，加速比是有上限的，不管你堆叠多少个 CPU 参与计算，都不能突破这个上限。

8.1.1 Gustafson's law 古斯塔夫森定律

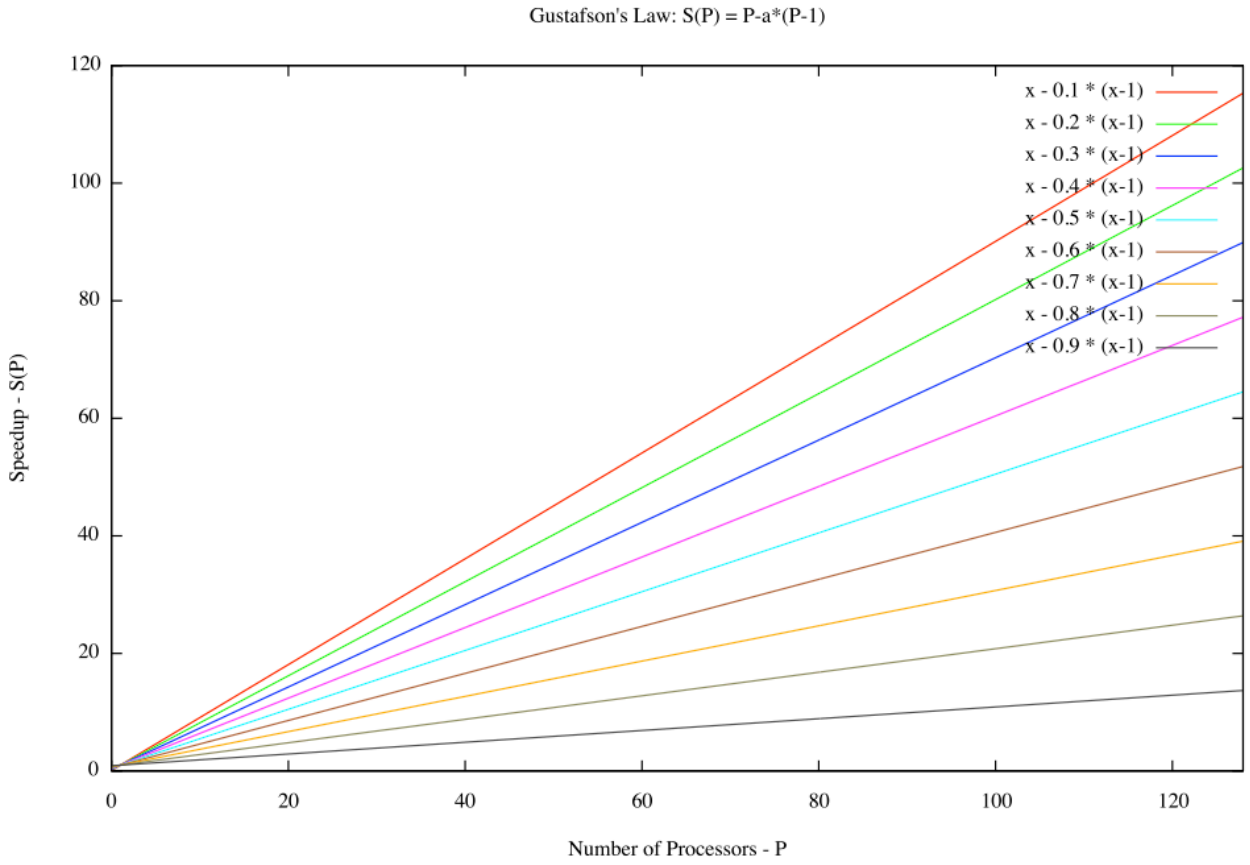


图 8.6: 古斯塔夫森定律

$$S_{\text{latency}}(s) = 1 - p + sp$$

图 8.7: 古斯塔夫森定律定义

古斯塔夫森定律强调的是：如果可被并行化的代码所占比例足够大，那么加速比就能随着 CPU 的数量线性增长。

8.1.2 JAVA 内存模型 JMM

JAVA 内存模型 Java Memory Model JMM

第 9 章　IPv6

9.1　基础知识

9.1.1　IPv6 格式

IPv6 二进位制下为 128 位长度。

9.1.2　问题

1. IP 地址转为 int 实现

2. DNS 域名解析中 A、AAAA、CNAME、MX、NS、TXT、SRV、SOA、PTR 各项记录的作用

https://blog.hackroad.com/operations-engineer/basics/13255.html

	standard font size		
command	10pt	11pt	12pt
\tiny	5pt	6pt	6pt
\scriptsize	7pt	8pt	8pt
\footnotesize	8pt	9pt	10pt
\small	9pt	10pt	11pt
\normalsize	10pt	11pt	12pt
\large	12pt	12pt	14pt
\Large	14pt	14pt	17pt
\LARGE	17pt	17pt	20pt
\huge	20pt	20pt	25pt
\Huge	25pt	25pt	25pt

`article, report, book und letter standard font size`